W3School Java & Java Web 教程

wizardforcel

Published with GitBook



目錄

W3School Java & Java Web教程

来源：

- Java教程
- JSP教程
- Servlet教程

整理：飞龙

Java 基础

Java 简介

Java是由Sun Microsystems公司于1995年5月推出的Java面向对象程序设计语言和Java平台的总称。由James Gosling和同事们共同研发，并在1995年正式推出。

Java分为三个体系：

- JavaSE（J2SE）（Java2 Platform Standard Edition，java平台标准版）
- JavaEE(J2EE)(Java 2 Platform,Enterprise Edition，java平台企业版)
- JavaME(J2ME)(Java 2 Platform Micro Edition，java平台微型版)。

2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名以取消其中的数字"2"：J2EE更名为Java EE, J2SE更名为Java SE，J2ME更名为Java ME。

主要特性

- **Java**语言是简单的：

  Java语言的语法与C语言和C++语言很接近，使得大多数程序员很容易学习和使用。另一方面，Java丢弃了C++中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，Java语言不使用指针，而是引用。并提供了自动的废料收集，使得程序员不必为内存管理而担忧。

- **Java**语言是面向对象的：

  Java语言提供类、接口和继承等原语，为了简单起见，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为implements）。Java语言全面支持动态绑定，而C++语言只对虚函数使用动态绑定。总之，Java语言是一个纯的面向对象程序设计语言。

- **Java**语言是分布式的：

  Java语言支持Internet应用的开发，在基本的Java应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括URL、URLConnection、Socket、ServerSocket等。Java的RMI（远程方法激活）机制也是开发分布式应用的重要手段。

- **Java**语言是健壮的：

  Java的强类型机制、异常处理、垃圾的自动收集等是Java程序健壮性的重要保证。对指针的丢弃是Java的明智选择。Java的安全检查机制使得Java更具健壮性。

- **Java**语言是安全的：

Java通常被用在网络环境中，为此，Java提供了一个安全机制以防恶意代码的攻击。除了Java语言具有的许多安全特性以外，Java对通过网络下载的类具有一个安全防范机制（类ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查，并提供安全管理机制（类SecurityManager）让Java应用设置安全哨兵。

- **Java语言是体系结构中立的：**

  Java程序（后缀为java的文件）在Java平台上被编译为体系结构中立的字节码格式（后缀为class的文件），然后可以在实现这个Java平台的任何系统中运行。这种途径适合于异构的网络环境和软件的分发。

- **Java语言是可移植的：**

  这种可移植性来源于体系结构中立性，另外，Java还严格规定了各个基本数据类型的长度。Java系统本身也具有很强的可移植性，Java编译器是用Java实现的，Java的运行环境是用ANSI C实现的。

- **Java语言是解释型的：**

  如前所述，Java程序在Java平台上被编译为字节码格式，然后可以在实现这个Java平台的任何系统中运行。在运行时，Java平台中的Java解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。

- **Java是高性能的：**

  与那些解释型的高级脚本语言相比，Java的确是高性能的。事实上，Java的运行速度随着JIT(Just-In-Time）编译器技术的发展越来越接近于C++。

- **Java语言是多线程的：**

  在Java语言中，线程是一种特殊的对象，它必须由Thread类或其子（孙）类来创建。通常有两种方法来创建线程：其一，使用型构为Thread(Runnable)的构造子将一个实现了Runnable接口的对象包装成一个线程，其二，从Thread类派生出子类并重写run方法，使用该子类创建的对象即为线程。值得注意的是Thread类已经实现了Runnable接口，因此，任何一个线程均有它的run方法，而run方法中包含了线程所要运行的代码。线程的活动由一组方法来控制。Java语言支持多个线程的同时执行，并提供多线程之间的同步机制（关键字为synchronized）。

- **Java语言是动态的：**

  Java语言的设计目标之一是适应于动态变化的环境。Java程序需要的类能够动态地被载入到运行环境，也可以通过网络来载入所需要的类。这也有利于软件的升级。另外，Java中的类有一个运行时刻的表示，能进行运行时刻的类型检查。

发展历史

- 1995年5月23日，Java语言诞生
- 1996年1月，第一个JDK-JDK1.0诞生
- 1996年4月，10个最主要的操作系统供应商申明将在其产品中嵌入JAVA技术
- 1996年9月，约8.3万个网页应用了JAVA技术来制作
- 1997年2月18日，JDK1.1发布
- 1997年4月2日，JavaOne会议召开，参与者逾一万人，创当时全球同类会议规模之纪录
- 1997年9月，JavaDeveloperConnection社区成员超过十万
- 1998年2月，JDK1.1被下载超过2,000,000次
- 1998年12月8日，JAVA2企业平台J2EE发布
- 1999年6月，SUN公司发布Java的三个版本：标准版（JavaSE,以前是J2SE）、企业版（JavaEE以前是J2EE）和微型版（JavaME，以前是J2ME）
- 2000年5月8日，JDK1.3发布
- 2000年5月29日，JDK1.4发布
- 2001年6月5日，NOKIA宣布，到2003年将出售1亿部支持Java的手机
- 2001年9月24日，J2EE1.3发布
- 2002年2月26日，J2SE1.4发布，自此Java的计算能力有了大幅提升
- 2004年9月30日18:00PM，J2SE1.5发布，成为Java语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5更名为Java SE 5.0
- 2005年6月，JavaOne大会召开，SUN公司公开Java SE 6。此时，Java的各种版本已经更名，以取消其中的数字"2"：J2EE更名为Java EE，J2SE更名为Java SE，J2ME更名为Java ME
- 2006年12月，SUN公司发布JRE6.0
- 2009年04月20日，甲骨文74亿美元收购Sun。取得java的版权。
- 2010年11月，由于甲骨文对于Java社区的不友善，因此Apache扬言将退出JCP[4]。
- 2011年7月28日，甲骨文发布java7.0的正式版。

Java开发工具

Java语言尽量保证系统内存在1G以上，其他工具如下所示：

- Linux 系统或者Windows 95/98/2000/XP，WIN 7/8系统
- Java JDK 7
- Notepad编辑器或者其他编辑器。
- IDE：Eclipse

安装好以上的工具后，我们就可以输出Java的第一个程序"Hello World！"

```
public class MyFirstJavaProgram {

    public static void main(String []args) {
       System.out.println("Hello World");
    }
}
```

在下一章节我们将介绍如何配置java开发环境。

Java开发环境配置

在本章节中我们将为大家介绍如何搭建Java开发环境。

window系统安装java

下载JDK

首先我们需要下载java开发工具包JDK，下载地址：http://www.oracle.com/technetwork/java/javase/downloads/index.html，点击如下下载按钮：

下载后JDK的安装根据提示进行，还有安装JDK的时候也会安装JRE，一并安装就可以了。

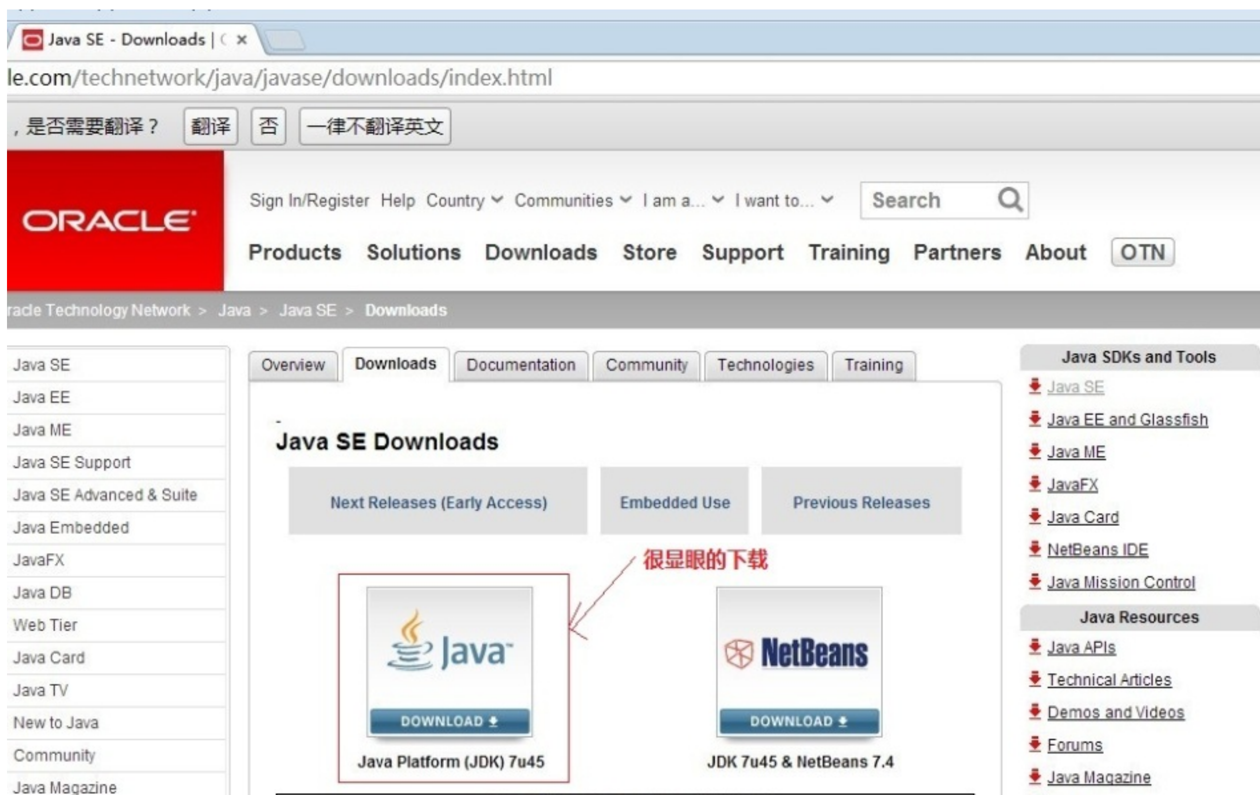
安装JDK，安装过程中可以自定义安装目录等信息，例如我们选择安装目录为C:\Program Files\Java\jdk1.7.0 。

配置环境变量

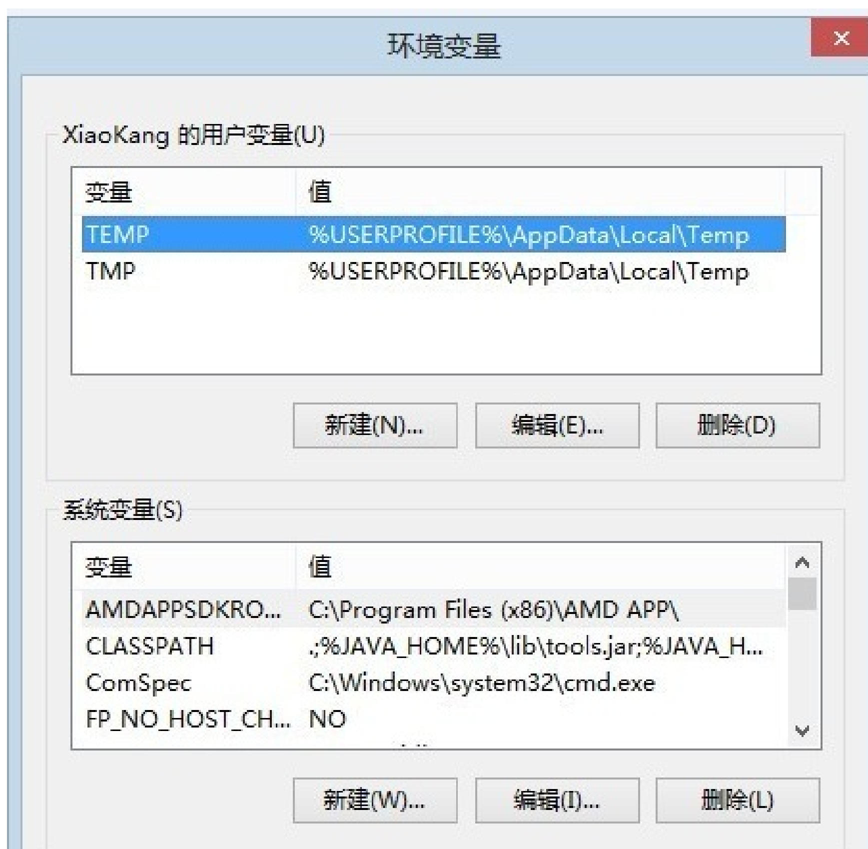
1.安装完成后，右击"我的电脑"，点击"属性"；

2.选择"高级"选项卡，点击"环境变量"；

然后就会出现如下图所示的画面



在"系统变量"中设置3项属性，JAVA_HOME,PATH,CLASSPATH(大小写无所谓),若已存在则点击"编辑"，不存在则点击"新建"。



变量设置

- 变量名：JAVA_HOME

- 变量值：C:\Program Files\Java\jdk1.7.0

//这里是你JDK的安装路径，可以更换

- 变量名：CLASSPATH
- 变量值：.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar; //记得前面有个"."

- 变量名：Path

- 变量值：%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

这是java的环境配置，配置完成后直接启动eclipse，它会自动完成java环境的配置。

测试JDK是否安装成功

1、"开始"->;"运行"，键入"cmd"；

2、键入命令"java -version"，"java"，"javac"几个命令，出现画面，说明环境变量配置成功；

```
C:\Users\XiaoKang>java
用法: java [-options] class [args...]
           (执行类)
   或  java [-options] -jar jarfile [args...]
           (执行 jar 文件)
其中选项包括:
    -d32          使用 32 位数据模型 (如果可用)
    -d64          使用 64 位数据模型 (如果可用)
    -server       选择 "server" VM
    -hotspot      是 "server" VM 的同义词 [已过时]
                  默认 VM 是 server.

    -cp <目录和 zip/jar 文件的类搜索路径>
    -classpath <目录和 zip/jar 文件的类搜索路径>
                  用 ; 分隔的目录, JAR 档案
                  和 ZIP 档案列表, 用于搜索类文件。
    -D<name>=<value>
                  设置系统属性
    -verbose[:class|gc|jni]
                  启用详细输出
```

Linux，UNIX，Solaris，FreeBSD环境变量设置

环境变量PATH应该设定为指向Java二进制文件安装的位置。如果设置遇到困难，请参考shell文档。

例如，假设你使用bash作为shell，你可以把下面的内容添加到你的 .bashrc文件结尾: export PATH=/path/to/java:$PATH

流行JAVA开发工具

正所谓工欲善其事必先利其器，我们在开发java语言过程中同样需要依款不错的开发工具，目前市场上的IDE很多，本文为大家推荐一下几款java开发工具：

- **Notepad++ :** Notepad++ 是在微软视窗环境之下的一个免费的代码编辑器，下载地址：http://notepad-plus-plus.org/
- **Netbeans:**开源免费的java IDE，下载地址：http://www.netbeans.org/index.html
- **Eclipse:**另一个免费开源的java IDE，下载地址：http://www.eclipse.org/

Java基础语法

一个Java程序可以认为是一系列对象的集合，而这些对象通过调用彼此的方法来协同工作。下面简要介绍下类、对象、方法和实例变量的概念。

- 对象：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- 类：类是一个模板，它描述一类对象的行为和状态。
- 方法：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。
- 实例变量：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

第一个Java程序

下面看一个简单的Java程序，它将打印字符串 *Hello World*

```
public class MyFirstJavaProgram {
   /* 第一个Java程序.
    * 它将打印字符串 Hello World
    */
    public static void main(String []args) {
       System.out.println("Hello World"); // 打印 Hello World
    }
}
```

下面将逐步介绍如何保存、编译以及运行这个程序：

- 打开Notepad，把上面的代码添加进去；
- 把文件名保存为：MyFirstJavaProgram.java；
- 打开cmd命令窗口，进入目标文件所在的位置，假设是C:\
- 在命令行窗口键入 javac MyFirstJavaProgram.java 按下enter键编译代码。如果代码没有错误，cmd命令提示符会进入下一行。（假设环境变量都设置好了）。
- 再键入java MyFirstJavaProgram 按下Enter键就可以运行程序了

你将会在窗口看到 Hello World

```
C : > javac MyFirstJavaProgram.java
C : > java MyFirstJavaProgram
Hello World
```

基本语法

编写Java程序时，应注意以下几点：

- 大小写敏感：Java是大小写敏感的，这就意味着标识符Hello与hello是不同的。
- 类名：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 MyFirstJavaClass 。
- 方法名：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。
- 源文件名：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记Java是大小写敏感的），文件名的后缀为.java。（如果文件名和类名不相同则会导致编译错误）。
- 主方法入口：所有的Java 程序由**public static void main(String args[])**方法开始执行。

Java标识符

Java所有的组成部分都需要名字。类名、变量名以及方法名都被称为标识符。

关于Java标识符，有以下几点需要注意：

- 所有的标识符都应该以字母（A-Z或者a-z）,美元符（$）、或者下划线（_）开始
- 首字符之后可以是任何字符的组合
- 关键字不能用作标识符
- 标识符是大小写敏感的
- 合法标识符举例：age、$salary、_value、__1_value
- 非法标识符举例：123abc、-salary

Java修饰符

像其他语言一样，Java可以使用修饰符来修饰类中方法和属性。主要有两类修饰符：

- 可访问修饰符 : default, public , protected, private
- 不可访问修饰符 : final, abstract, strictfp

在后面的章节中我们会深入讨论Java修饰符。

Java变量

Java中主要有如下几种类型的变量

- 局部变量
- 类变量（静态变量）
- 成员变量（非静态变量）

Java数组

数组是储存在堆上的对象，可以保存多个同类型变量。在后面的章节中，我们将会学到如何声明、构造以及初始化一个数组。

Java枚举

Java 5.0引入了枚举，枚举限制变量只能是预先设定好的值。使用枚举可以减少代码中的bug。

例如，我们为果汁店设计一个程序，它将限制果汁为小杯、中杯、大杯。这就意味着它不允许顾客点除了这三种尺寸外的果汁。

实例

```
class FreshJuice {
   enum FreshJuiceSize{ SMALL, MEDUIM, LARGE }
   FreshJuiceSize size;
}

public class FreshJuiceTest {
   public static void main(String args[]){
      FreshJuice juice = new FreshJuice();
      juice.size = FreshJuice. FreshJuiceSize.MEDUIM ;
   }
}
```

注意：枚举可以单独声明或者声明在类里面。方法、变量、构造函数也可以在枚举中定义。

Java关键字

下面列出了Java保留字。这些保留字不能用于常量、变量、和任何标识符的名称。

关键字	描述
abstract	抽象方法，抽象类的修饰符
assert	断言条件是否满足
boolean	布尔数据类型
break	跳出循环或者label代码段
byte	8-bit 有符号数据类型
case	switch语句的一个条件
catch	和try搭配扑捉异常信息
char	16-bit Unicode字符数据类型
class	定义类

const	未使用
continue	不执行循环体剩余部分
default	switch语句中的默认分支
do	循环语句，循环体至少会执行一次
double	64-bit双精度浮点数
else	if条件不成立时执行的分支
enum	枚举类型
extends	表示一个类是另一个类的子类
final	表示一个值在初始化之后就不能再改变了 表示方法不能被重写，或者一个类不能有子类
finally	为了完成执行的代码而设计的，主要是为了程序的健壮性和完整性，无论有没有异常发生都执行代码。
float	32-bit单精度浮点数
for	for循环语句
goto	未使用
if	条件语句
implements	表示一个类实现了接口
import	导入类
instanceof	测试一个对象是否是某个类的实例
int	32位整型数
interface	接口，一种抽象的类型，仅有方法和常量的定义
long	64位整型数
native	表示方法用非java代码实现
new	分配新的类实例
package	一系列相关类组成一个包
private	表示私有字段，或者方法等，只能从类内部访问
protected	表示字段只能通过类或者其子类访问 子类或者在同一个包内的其他类
public	表示共有属性或者方法
return	方法返回值
short	16位数字
static	表示在类级别定义，所有实例共享的
strictfp	浮点数比较使用严格的规则

super	表示基类
switch	选择语句
synchronized	表示同一时间只能由一个线程访问的代码块
this	表示调用当前实例 或者调用另一个构造函数
throw	抛出异常
throws	定义方法可能抛出的异常
transient	修饰不要序列化的字段
try	表示代码块要做异常处理或者和finally配合表示是否抛出异常都执行finally中的代码
void	标记方法不返回任何值
volatile	标记字段可能会被多个线程同时访问，而不做同步
while	while循环

Java注释

类似于C/C++，Java也支持单行以及多行注释。注释中的字符将被Java编译器忽略。

```
public class MyFirstJavaProgram{
   /* 这是第一个Java程序
    *它将打印Hello World
    * 这是一个多行注释的示例
    */
    public static void main(String []args){
       // 这是单行注释的示例
       /* 这个也是单行注释的示例 */
       System.out.println("Hello World");
    }
}
```

Java 空行

空白行，或者有注释的的行，Java编译器都会忽略掉。

继承

在Java中，一个类可以由其他类派生。如果你要创建一个类，而且已经存在一个类具有你所需要的属性或方法，那么你可以将新创建的类继承该类。

利用继承的方法，可以重用已存在类的方法和属性，而不用重写这些代码。被继承的类称为超类（super class），派生类称为子类（subclass）。

接口

在Java中，接口可理解为对象间相互通信的协议。接口在继承中扮演着很重要的角色。

接口只定义派生要用到的方法，但是方法的具体实现完全取决于派生类。

下一节介绍Java编程中的类和对象。之后你将会对Java中的类和对象有更清楚的认识。

Java对象和类

Java作为一种面向对象语言。支持以下基本概念：

- 多态
- 继承
- 封装
- 抽象
- 类
- 对象
- 实例
- 方法
- 消息解析

本节我们重点研究对象和类的概念。

- 对象：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- 类：类是一个模板，它描述一类对象的行为和状态。

Java中的对象

现在让我们深入了解什么是对象。看看周围真实的世界，会发现身边有很多对象，车，狗，人等等。所有这些对象都有自己的状态和行为。

拿一条狗来举例，它的状态有：名字、品种、颜色，行为有：叫、摇尾巴和跑。

对比现实对象和软件对象，它们之间十分相似。

软件对象也有状态和行为。软件对象的状态就是属性，行为通过方法体现。

在软件开发中，方法操作对象内部状态的改变，对象的相互调用也是通过方法来完成。

Java中的类

类可以看成是创建Java对象的模板。

通过下面一个简单的类来理解下Java中类的定义：

```
public class Dog{
   String breed;
   int age;
   String color;
   void barking(){
   }

   void hungry(){
   }

   void sleeping(){
   }
}
```

一个类可以包含以下类型变量：

- 局部变量：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
- 成员变量：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
- 类变量：类变量也声明在类中，方法体之外，但必须声明为static类型。

一个类可以拥有多个方法，在上面的例子中：barking()、hungry()和sleeping()都是Dog类的方法。

构造方法

每个类都有构造方法。如果没有显式地为类定义构造方法，Java编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

下面是一个构造方法示例：

```
public class Puppy{
   public Puppy(){
   }

   public Puppy(String name){
      // 这个构造器仅有一个参数：name
   }
}
```

创建对象

对象是根据类创建的。在Java中，使用关键字new来创建一个新的对象。创建对象需要以下三步：

- 声明：声明一个对象，包括对象名称和对象类型。

- 实例化：使用关键字new来创建一个对象。
- 初始化：使用new创建对象时，会调用构造方法初始化对象。

下面是一个创建对象的例子：

```
public class Puppy{
   public Puppy(String name){
      //这个构造器仅有一个参数：name
      System.out.println("Passed Name is :" + name );
   }
   public static void main(String []args){
      // 下面的语句将创建一个Puppy对象
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

编译并运行上面的程序，会打印出下面的结果：

```
Passed Name is :tommy
```

访问实例变量和方法

通过已创建的对象来访问成员变量和成员方法，如下所示：

```
/* 实例化对象 */
ObjectReference = new Constructor();
/* 访问其中的变量 */
ObjectReference.variableName;
/* 访问类中的方法 */
ObjectReference.MethodName();
```

实例

下面的例子展示如何访问实例变量和调用成员方法：

```
public class Puppy{
   int puppyAge;
   public Puppy(String name){
      // 这个构造器仅有一个参数：name
      System.out.println("Passed Name is :" + name );
   }

   public void setAge( int age ){
       puppyAge = age;
   }

   public int getAge( ){
       System.out.println("Puppy's age is :" + puppyAge );
       return puppyAge;
   }

   public static void main(String []args){
      /* 创建对象 */
      Puppy myPuppy = new Puppy( "tommy" );
      /* 通过方法来设定age */
      myPuppy.setAge( 2 );
      /* 调用另一个方法获取age */
      myPuppy.getAge( );
      /*你也可以像下面这样访问成员变量 */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

编译并运行上面的程序，产生如下结果：

```
Passed Name is :tommy
Puppy's age is :2
Variable Value :2
```

源文件声明规则

在本节的最后部分，我们将学习源文件的声明规则。当在一个源文件中定义多个类，并且还有import语句和package语句时，要特别注意这些规则。

- 一个源文件中只能有一个public类
- 一个源文件可以有多个非public类
- 源文件的名称应该和public类的类名保持一致。例如：源文件中public类的类名是Employee，那么源文件应该命名为Employee.java。
- 如果一个类定义在某个包中，那么package语句应该在源文件的首行。
- 如果源文件包含import语句，那么应该放在package语句和类定义之间。如果没有package语句，那么import语句应该在源文件中最前面。
- import语句和package语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明。

类有若干种访问级别，并且类也分不同的类型：抽象类和final类等。这些将在访问控制章节介绍。

除了上面提到的几种类型，Java还有一些特殊的类，如：内部类、匿名类。

Java包

包主要用来对类和接口进行分类。当开发Java程序时，可能编写成百上千的类，因此很有必要对类和接口进行分类。

Import语句

在Java中，如果给出一个完整的限定名，包括包名、类名，那么Java编译器就可以很容易地定位到源代码或者类。Import语句就是用来提供一个合理的路径，使得编译器可以找到某个类。

例如，下面的命令行将会命令编译器载入java_installation/java/io路径下的所有类

```
import java.io.*;
```

一个简单的例子

在该例子中，我们创建两个类：Employee和EmployeeTest。

首先打开文本编辑器，把下面的代码粘贴进去。注意将文件保存为Employee.java。

Employee类有四个成员变量：name、age、designation和salary。该类显式声明了一个构造方法，该方法只有一个参数。

```
import java.io.*;
public class Employee{
   String name;
   int age;
   String designation;
   double salary;
   // Employee 类的构造器
   public Employee(String name){
      this.name = name;
   }
   // 设置age的值
   public void empAge(int empAge){
      age =  empAge;
   }
   /* 设置designation的值*/
   public void empDesignation(String empDesig){
      designation = empDesig;
   }
   /* 设置salary的值*/
   public void empSalary(double empSalary){
      salary = empSalary;
   }
   /* 打印信息 */
   public void printEmployee(){
      System.out.println("Name:"+ name );
      System.out.println("Age:" + age );
      System.out.println("Designation:" + designation );
      System.out.println("Salary:" + salary);
   }
}
```

程序都是从main方法开始执行。为了能运行这个程序，必须包含main方法并且创建一个实例对象。

下面给出EmployeeTest类，该类实例化2个Employee类的实例，并调用方法设置变量的值。

将下面的代码保存在EmployeeTest.java文件中。

```
import java.io.*;
public class EmployeeTest{

   public static void main(String args[]){
      /* 使用构造器创建两个对象 */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");

      // 调用这两个对象的成员方法
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
      empTwo.empSalary(500);
      empTwo.printEmployee();
   }
}
```

编译这两个文件并且运行EmployeeTest类，可以看到如下结果：

```
C :> javac Employee.java
C :> vi EmployeeTest.java
C :> javac  EmployeeTest.java
C :> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

Java基本数据类型

变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。

内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。

因此，通过定义不同类型的变量，可以在内存中储存整数、小数或者字符。

Java的两大数据类型:

- 内置数据类型
- 引用数据类型

内置数据类型

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

byte：

- byte数据类型是8位、有符号的，以二进制补码表示的整数；
- 最小值是-128（-2^7）；
- 最大值是127（2^7-1）；
- 默认值是0；
- byte类型用在大型数组中节约空间，主要代替整数，因为byte变量占用的空间只有int类型的四分之一；
- 例子：byte a = 100，byte b = -50。

short：

- short数据类型是16位、有符号的以二进制补码表示的整数
- 最小值是-32768（-2^15）；
- 最大值是32767（2^15 - 1）；
- Short数据类型也可以像byte那样节省空间。一个short变量是int型变量所占空间的二分之一；
- 默认值是0；
- 例子：short s = 1000，short r = -20000。

int：

- int数据类型是32位、有符号的以二进制补码表示的整数；
- 最小值是-2,147,483,648（-2^31）；
- 最大值是2,147,485,647（2^31 - 1）；

- 一般地整型变量默认为int类型；
- 默认值是0；
- 例子：int a = 100000, int b = -200000。

long：

- long数据类型是64位、有符号的以二进制补码表示的整数；
- 最小值是-9,223,372,036,854,775,808（-2^63）；
- 最大值是9,223,372,036,854,775,807（2^63 -1）；
- 这种类型主要使用在需要比较大整数的系统上；
- 默认值是0L；
- 例子： long a = 100000L，int b = -200000L。

float：

- float数据类型是单精度、32位、符合IEEE 754标准的浮点数；
- float在储存大型浮点数组的时候可节省内存空间；
- 默认值是0.0f；
- 浮点数不能用来表示精确的值，如货币；
- 例子：float f1 = 234.5f。

double：

- double数据类型是双精度、64位、符合IEEE 754标准的浮点数；
- 浮点数的默认类型为double类型；
- double类型同样不能表示精确的值，如货币；
- 默认值是0.0f；
- 例子：double d1 = 123.4。

boolean：

- boolean数据类型表示一位的信息；
- 只有两个取值：true和false；
- 这种类型只作为一种标志来记录true/false情况；
- 默认值是false；
- 例子：boolean one = true。

char：

- char类型是一个单一的16位Unicode字符；
- 最小值是'\u0000'（即为0）；
- 最大值是'\uffff'（即为65,535）；
- char数据类型可以储存任何字符；
- 例子：char letter = 'A'。

实例

对于数值类型的基本类型的取值范围，我们无需强制去记忆，因为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

```
public class PrimitiveTypeTest {
    public static void main(String[] args) {
        // byte
        System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);
        System.out.println("包装类：java.lang.Byte");
        System.out.println("最小值：Byte.MIN_VALUE=" + Byte.MIN_VALUE);
        System.out.println("最大值：Byte.MAX_VALUE=" + Byte.MAX_VALUE);
        System.out.println();

        // short
        System.out.println("基本类型：short 二进制位数：" + Short.SIZE);
        System.out.println("包装类：java.lang.Short");
        System.out.println("最小值：Short.MIN_VALUE=" + Short.MIN_VALUE);
        System.out.println("最大值：Short.MAX_VALUE=" + Short.MAX_VALUE);
        System.out.println();

        // int
        System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);
        System.out.println("包装类：java.lang.Integer");
        System.out.println("最小值：Integer.MIN_VALUE=" + Integer.MIN_VALUE);
        System.out.println("最大值：Integer.MAX_VALUE=" + Integer.MAX_VALUE);
        System.out.println();

        // long
        System.out.println("基本类型：long 二进制位数：" + Long.SIZE);
        System.out.println("包装类：java.lang.Long");
        System.out.println("最小值：Long.MIN_VALUE=" + Long.MIN_VALUE);
        System.out.println("最大值：Long.MAX_VALUE=" + Long.MAX_VALUE);
        System.out.println();

        // float
        System.out.println("基本类型：float 二进制位数：" + Float.SIZE);
        System.out.println("包装类：java.lang.Float");
        System.out.println("最小值：Float.MIN_VALUE=" + Float.MIN_VALUE);
        System.out.println("最大值：Float.MAX_VALUE=" + Float.MAX_VALUE);
        System.out.println();

        // double
        System.out.println("基本类型：double 二进制位数：" + Double.SIZE);
        System.out.println("包装类：java.lang.Double");
        System.out.println("最小值：Double.MIN_VALUE=" + Double.MIN_VALUE);
        System.out.println("最大值：Double.MAX_VALUE=" + Double.MAX_VALUE);
        System.out.println();

        // char
        System.out.println("基本类型：char 二进制位数：" + Character.SIZE);
        System.out.println("包装类：java.lang.Character");
        // 以数值形式而不是字符形式将Character.MIN_VALUE输出到控制台
        System.out.println("最小值：Character.MIN_VALUE="
                + (int) Character.MIN_VALUE);
        // 以数值形式而不是字符形式将Character.MAX_VALUE输出到控制台
        System.out.println("最大值：Character.MAX_VALUE="
                + (int) Character.MAX_VALUE);
    }
}
```

编译以上代码输出结果如下所示：

```
基本类型：byte 二进制位数：8
包装类：java.lang.Byte
最小值：Byte.MIN_VALUE=-128
最大值：Byte.MAX_VALUE=127

基本类型：short 二进制位数：16
包装类：java.lang.Short
最小值：Short.MIN_VALUE=-32768
最大值：Short.MAX_VALUE=32767

基本类型：int 二进制位数：32
包装类：java.lang.Integer
最小值：Integer.MIN_VALUE=-2147483648
最大值：Integer.MAX_VALUE=2147483647

基本类型：long 二进制位数：64
包装类：java.lang.Long
最小值：Long.MIN_VALUE=-9223372036854775808
最大值：Long.MAX_VALUE=9223372036854775807

基本类型：float 二进制位数：32
包装类：java.lang.Float
最小值：Float.MIN_VALUE=1.4E-45
最大值：Float.MAX_VALUE=3.4028235E38

基本类型：double 二进制位数：64
包装类：java.lang.Double
最小值：Double.MIN_VALUE=4.9E-324
最大值：Double.MAX_VALUE=1.7976931348623157E308

基本类型：char 二进制位数：16
包装类：java.lang.Character
最小值：Character.MIN_VALUE=0
最大值：Character.MAX_VALUE=65535
```

Float和Double的最小值和最大值都是以科学记数法的形式输出的，结尾的"E+数字"表示E之前的数字要乘以10的多少倍。比如3.14E3就是3.14×1000=3140，3.14E-3就是3.14/1000=0.00314。

实际上，JAVA中还存在另外一种基本类型void，它也有对应的包装类 java.lang.Void，不过我们无法直接对它们进行操作。

引用类型

- 引用类型变量由类的构造函数创建，可以使用它们访问所引用的对象。这些变量在声明时被指定为一个特定的类型，比如Employee、Pubby等。变量一旦声明后，类型就不能被改变了。
- 对象、数组都是引用数据类型。
- 所有引用类型的默认值都是null。
- 一个引用变量可以用来引用与任何与之兼容的类型。
- 例子：Animal animal = new Animal(“giraffe”)。

Java常量

常量就是一个固定值。它们不需要计算，直接代表相应的值。

常量指不能改变的量。 在Java中用final标志，声明方式和变量类似：

```
final double PI = 3.1415927;
```

虽然常量名也可以用小写，但为了便于识别，通常使用大写字母表示常量。

字面量可以赋给任何内置类型的变量。例如：

```
byte a = 68;
char a = 'A'
```

byte、int、long、和short都可以用十进制、16进制以及8进制的方式来表示。

当使用常量的时候，前缀o表明是8进制，而前缀0x代表16进制。例如：

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

和其他语言一样，Java的字符串常量也是包含在两个引号之间的字符序列。下面是字符串型字面量的例子：

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

字符串常量和字符常量都可以包含任何Unicode字符。例如：

```
char a = '\u0001';
String a = "\u0001";
```

Java语言支持一些特殊的转义字符序列。

符号	字符含义
\n	换行 (0x0a)
\r	回车 (0x0d)
\f	换页符(0x0c)
\b	退格 (0x08)
\s	空格 (0x20)
\t	制表符
\"	双引号
\'	单引号
\	反斜杠
\ddd	八进制字符 (ddd)
\uxxxx	16进制Unicode字符 (xxxx)

这一节讲解了Java的基本数据类型。下一节将探讨不同的变量类型以及它们的用法。

Java变量类型

在Java语言中，所有的变量在使用前必须声明。声明变量的基本格式如下：

```
type identifier [ = value][, identifier [= value] ...] ;
```

格式说明：type为Java数据类型。identifier是变量名。可以使用逗号隔开来声明多个同类型变量。

以下列出了一些变量的声明实例。注意有些包含了初始化过程。

```
int a, b, c;         // 声明三个int型整数：a、 b、c。
int d = 3, e, f = 5; // d声明三个整数并赋予初值。
byte z = 22;         // 声明并初始化z。
double pi = 3.14159; // 声明了pi。
char x = 'x';        // 变量x的值是字符'x'。
```

Java语言支持的变量类型有：

- 局部变量
- 成员变量
- 类变量

Java局部变量

- 局部变量声明在方法、构造方法或者语句块中；
- 局部变量在方法、构造方法、或者语句块被执行的时候创建，当它们执行完成后，变量将会被销毁；
- 访问修饰符不能用于局部变量；
- 局部变量只在声明它的方法、构造方法或者语句块中可见；
- 局部变量是在栈上分配的。
- 局部变量没有默认值，所以局部变量量被声明后，必须经过初始化，才可以使用。

实例1

在以下实例中age是一个局部变量。定义在pubAge()方法中，它的作用域就限制在这个方法中。

```
public class Test{
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

以上实例编译运行结果如下：

```
Puppy age is: 7
```

实例2

在下面的例子中age变量没有初始化，所以在编译时出错。

```
public class Test{
   public void pupAge(){
      int age;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

以上实例编译运行结果如下:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
         ^
1 error
```

实例变量

- 实例变量声明在一个类中，但在方法、构造方法和语句块之外；
- 当一个对象被实例化之后，每个实例变量的值就跟着确定；
- 实例变量在对象创建的时候创建，在对象被销毁的时候销毁；
- 实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息；
- 实例变量可以声明在使用前或者使用后；
- 访问修饰符可以修饰实例变量；

- 实例变量对于类中的方法、构造方法或者语句块是可见的。一般情况下应该把实例变量设为私有。通过使用访问修饰符可以使实例变量对子类可见；
- 实例变量具有默认值。数值型变量的默认值是0，布尔型变量的默认值是false，引用类型变量的默认值是null。变量的值可以在声明时指定，也可以在构造方法中指定；
- 实例变量可以直接通过变量名访问。但在静态方法以及其他类中，就应该使用完全限定名：ObejectReference.VariableName。

实例：

```
import java.io.*;
public class Employee{
   // 这个成员变量对子类可见
   public String name;
   // 私有变量，仅在该类可见
   private double salary;
   //在构造器中对name赋值
   public Employee (String empName){
      name = empName;
   }
   //设定salary的值
   public void setSalary(double empSal){
      salary = empSal;
   }
   // 打印信息
   public void printEmp(){
      System.out.println("name  : " + name );
      System.out.println("salary :" + salary);
   }

   public static void main(String args[]){
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

以上实例编译运行结果如下:

```
name  : Ransika
salary :1000.0
```

类变量（静态变量）

- 类变量也称为静态变量，在类中以static关键字声明，但必须在方法构造方法和语句块之外。
- 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。
- 静态变量除了被声明为常量外很少使用。常量是指声明为publc/private，final和static类型的变量。常量初始化后不可改变。
- 静态变量储存在静态存储区。经常被声明为常量，很少单独使用static声明变量。
- 静态变量在程序开始时创建，在程序结束时销毁。
- 与实例变量具有相似的可见性。但为了对类的使用者可见，大多数静态变量声明为public类型。

- 默认值和实例变量相似。数值型变量默认值是0，布尔型默认值是false，引用类型默认值是null。变量的值可以在声明的时候指定，也可以在构造方法中指定。此外，静态变量还可以在静态语句块中初始化。
- 静态变量可以通过：*ClassName.VariableName*的方式访问。
- 类变量被声明为public static final类型时，类变量名称必须使用大写字母。如果静态变量不是public和final类型，其命名方式与实例变量以及局部变量的命名方式一致。

实例：

```
import java.io.*;
public class Employee{
   //salary是静态的私有变量
   private static double salary;
   // DEPARTMENT是一个常量
   public static final String DEPARTMENT = "Development ";
   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

以上实例编译运行结果如下:

```
Development average salary:1000
```

注意：如果其他类想要访问该变量，可以这样访问：Employee.DEPARTMENT。

本章节中我们学习了Java的变量类型，下一章节中我们将介绍Java修饰符的使用。

Java修饰符

Java语言提供了很多修饰符，主要分为以下两类：

- 访问修饰符
- 非访问修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端。我们通过下面的例子来说明：

```
public class className {
   // ...
}
private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
public static void main(String[] arguments) {
   // 方法体
}
```

访问控制修饰符

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java支持4种不同的访问权限。

默认的，也称为default，在同一包内可见，不使用任何修饰符。

私有的，以private修饰符指定，在同一类内可见。

共有的，以public修饰符指定，对所有类可见。

受保护的，以protected修饰符指定，对同一包内的类和所有子类可见。

默认访问修饰符-不使用任何关键字

使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为public static final,而接口里的方法默认情况下访问权限为public。

实例：

如下例所示，变量和方法的声明可以不使用任何修饰符。

```
String version = "1.5.1";
boolean processOrder() {
   return true;
}
```

私有访问修饰符-private

私有访问修饰符是最严格的访问级别，所以被声明为private的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为private。

声明为私有访问类型的变量只能通过类中公共的getter方法被外部类访问。

Private访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。

下面的类使用了私有访问修饰符：

```
public class Logger {
   private String format;
   public String getFormat() {
      return this.format;
   }
   public void setFormat(String format) {
      this.format = format;
   }
}
```

实例中，Logger类中的format变量为私有变量，所以其他类不能直接得到和设置该变量的值。为了使其他类能够操作该变量，定义了两个public方法：getFormat() （返回format的值）和setFormat(String)（设置format的值）

公有访问修饰符-public

被声明为public的类、方法、构造方法和接口能够被任何其他类访问。

如果几个相互访问的public类分布在不用的包中，则需要导入相应public类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

以下函数使用了公有访问控制：

```
public static void main(String[] arguments) {
   // ...
}
```

Java程序的main() 方法必须设置成公有的，否则，Java解释器将不能运行该类。

受保护的访问修饰符-protected

被声明为protected的变量、方法和构造器能被同一个包中的任何其他类访问，也能够被不同包中的子类访问。

Protected访问修饰符不能修饰类和接口，方法和成员变量能够声明为protected，但是接口的成员变量和成员方法不能声明为protected。

子类能访问Protected修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

下面的父类使用了protected访问修饰符，子类重载了父类的openSpeaker()方法。

```
class AudioPlayer {
   protected boolean openSpeaker(Speaker sp) {
      // 实现细节
   }
}

class StreamingAudioPlayer {
   boolean openSpeaker(Speaker sp) {
      // 实现细节
   }
}
```

如果把openSpeaker()方法声明为private，那么除了AudioPlayer之外的类将不能访问该方法。如果把openSpeaker()声明为public，那么所有的类都能够访问该方法。如果我们只想让该方法对其所在类的子类可见，则将该方法声明为protected。

访问控制和继承

请注意以下方法继承的规则：

- 父类中声明为public的方法在子类中也必须为public。
- 父类中声明为protected的方法在子类中要么声明为protected，要么声明为public。不能声明为private。
- 父类中默认修饰符声明的方法，能够在子类中声明为private。
- 父类中声明为private的方法，不能够被继承。

非访问修饰符

为了实现一些其他的功能，Java也提供了许多非访问修饰符。

static修饰符，用来创建类方法和类变量。

Final修饰符，用来修饰类、方法和变量，final修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。

Abstract修饰符，用来创建抽象类和抽象方法。

Synchronized和volatile修饰符，主要用于线程的编程。

Static修饰符

- 静态变量：

  Static关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被成为类变量。局部变量能被声明为static变量。

- 静态方法：

  Static关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

对类变量和方法的访问可以直接使用classname.variablename和classname.methodname的方式访问。

如下例所示，static修饰符用来创建类方法和类变量。

```
public class InstanceCounter {
   private static int numInstances = 0;
   protected static int getCount() {
      return numInstances;
   }

   private static void addInstance() {
      numInstances++;
   }

   InstanceCounter() {
      InstanceCounter.addInstance();
   }

   public static void main(String[] arguments) {
      System.out.println("Starting with " +
      InstanceCounter.getCount() + " instances");
      for (int i = 0; i < 500; ++i){
         new InstanceCounter();
          }
      System.out.println("Created " +
      InstanceCounter.getCount() + " instances");
   }
}
```

以上实例运行编辑结果如下:

```
Started with 0 instances
Created 500 instances
```

Final修饰符

Final变量：

Final变量能被显式地初始化并且只能初始化一次。被声明为final的对象的引用不能指向不同的对象。但是final对象里的数据可以被改变。也就是说final对象的引用不能改变，但是里面的值可以改变。

Final修饰符通常和static修饰符一起使用来创建类常量。

实例:

```
public class Test{
  final int value = 10;
  // 下面是声明常量的实例
  public static final int BOXWIDTH = 6;
  static final String TITLE = "Manager";

  public void changeValue(){
     value = 12; //将输出一个错误
  }
}
```

Final方法

类中的Final方法可以被子类继承，但是不能被子类修改。

声明final方法的主要目的是防止该方法的内容被修改。

如下所示，使用final修饰符声明方法。

```
public class Test{
    public final void changeName(){
       // 方法体
    }
}
```

Final类

Final类不能被继承，没有类能够继承final类的任何特性。

实例：

```
public final class Test {
   // 类体
}
```

Abstract修饰符

抽象类：

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。

一个类不能同时被abstract和final修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

实例：

```
abstract class Caravan{
   private double price;
   private String model;
   private String year;
   public abstract void goFast(); //抽象方法
   public abstract void changeColor();
}
```

抽象方法

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。抽象方法不能被声明成final和strict。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

抽象方法的声明以分号结尾，例如：public abstract sample();

实例：

```
public abstract class SuperClass{
    abstract void m(); //抽象方法
}

class SubClass extends SuperClass{
     //实现抽象方法
      void m(){
          .........
      }
}
```

Synchronized修饰符

Synchronized关键字声明的方法同一时间只能被一个线程访问。Synchronized修饰符可以应用于四个访问修饰符。

实例：

```
public synchronized void showDetails(){
.......
}
```

Transient修饰符

序列化的对象包含被transient修饰的实例变量时，java虚拟机(JVM)跳过该特定的变量。

该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

实例：

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

volatile修饰符

Volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。一个volatile对象引用可能是null。

实例：

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;
    public void run()
    {
        active = true;
        while (active) // line 1
        {
            // 代码
        }
    }
    public void stop()
    {
        active = false; // line 2
    }
}
```

一般地，在一个线程中调用run()方法，在另一个线程中调用stop()方法。如果line 1中的active位于缓冲区的值被使用，那么当把line 2中的active设置成false时，循环也不会停止。

Java运算符

计算机的最基本用途之一就是执行数学运算，作为一门计算机语言，Java也提供了一套丰富的运算符来操纵变量。我们可以把运算符分成以下几组：

- 算术运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

算术运算符

算术运算符用在数学表达式中，它们的作用和在数学中的作用一样。下表列出了所有的算术运算符。

表格中的实例假设整数变量A的值为10，变量B的值为20：

操作符	描述	例子
+	加法 - 相加运算符两侧的值	A + B等于30
-	减法 - 左操作数减去右操作数	A – B等于-10
*	乘法 - 相乘操作符两侧的值	A * B等于200
/	除法 - 左操作数除以右操作数	B / A等于2
%	取模 - 右操作数除左操作数的余数	B%A等于0
+ +	自增 - 操作数的值增加1	B + +等于21
-	自减 - 操作数的值减少1	B - -等于19

实例

下面的简单示例程序演示了算术运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 25;
     int d = 25;
     System.out.println("a + b = " + (a + b) );
     System.out.println("a - b = " + (a - b) );
     System.out.println("a * b = " + (a * b) );
     System.out.println("b / a = " + (b / a) );
     System.out.println("b % a = " + (b % a) );
     System.out.println("c % a = " + (c % a) );
     System.out.println("a++   = " +  (a++) );
     System.out.println("b--   = " +  (a--) );
     // Check the difference in d++ and ++d
     System.out.println("d++   = " +  (d++) );
     System.out.println("++d   = " +  (++d) );
  }
}
```

以上实例编译运行结果如下：

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

关系运算符

下表为Java支持的关系运算符

表格中的实例整数变量A的值为10，变量B的值为20：

运算符	描述	例子
==	检查如果两个操作数的值是否相等，如果相等则条件为真。	（A == B）为假(非真)。
=!	检查如果两个操作数的值是否相等，如果值不相等则条件为真。	（A =! B）为真。
>	检查左操作数的值是否大于右操作数的值，如果是那么条件为真。	（A > B）非真。
<	检查左操作数的值是否小于右操作数的值，如果是那么条件为真。	（A < B）为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。	（A >= B）为假。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。	（A <= B）为真。

实例

下面的简单示例程序演示了关系运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     System.out.println("a == b = " + (a == b) );
     System.out.println("a != b = " + (a != b) );
     System.out.println("a > b = " + (a > b) );
     System.out.println("a < b = " + (a < b) );
     System.out.println("b >= a = " + (b >= a) );
     System.out.println("b <= a = " + (b <= a) );
  }
}
```

以上实例编译运行结果如下：

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

位运算符

Java定义了位运算符，应用于整数类型(int)，长整型(long)，短整型(short)，字符型(char)，和字节型(byte)等类型。

位运算符作用在所有的位上，并且按位运算。假设a = 60，和b = 13;它们的二进制格式表示将如下：

```
A = 0011 1100
B = 0000 1101
-----------------
A&b = 0000 1100
A | B = 0011 1101
^ B = 0011 0001
~A= 1100 0011
```

下表列出了位运算符的基本运算,假设整数变量A的值为60和变量B的值为13：

操作符	描述	例子
&	按位与操作符，当且仅当两个操作数的某一位都非0时候结果的该位才为1。	（A&B），得到12，即0000 1100
|	按位或操作符，只要两个操作数的某一位有一个非0时候结果的该位就为1。	（A | B）得到61，即 0011 1101
^	按位异或操作符，两个操作数的某一位不相同时候结果的该位就为1。	（A ^ B）得到49，即 0011 0001
?	按位补运算符翻转操作数的每一位。	（?A）得到-60，即1100 0011
<<	按位左移运算符。左操作数按位左移右操作数指定的位数。	A << 2得到240，即1111 0000
>>	按位右移运算符。左操作数按位右移右操作数指定的位数。	A >> 2得到15即1111
>>>	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。	A>>>2得到15即0000 1111

实例

下面的简单示例程序演示了位运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {
  public static void main(String args[]) {
     int a = 60; /* 60 = 0011 1100 */
     int b = 13; /* 13 = 0000 1101 */
     int c = 0;
     c = a & b;       /* 12 = 0000 1100 */
     System.out.println("a & b = " + c );

     c = a | b;       /* 61 = 0011 1101 */
     System.out.println("a | b = " + c );

     c = a ^ b;       /* 49 = 0011 0001 */
     System.out.println("a ^ b = " + c );

     c = ~a;          /*-61 = 1100 0011 */
     System.out.println("~a = " + c );

     c = a << 2;     /* 240 = 1111 0000 */
     System.out.println("a << 2 = " + c );

     c = a >> 2;     /* 215 = 1111 */
     System.out.println("a >> 2  = " + c );

     c = a >>> 2;     /* 215 = 0000 1111 */
     System.out.println("a >>> 2 = " + c );
  }
}
```

以上实例编译运行结果如下：

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

逻辑运算符

下表列出了逻辑运算符的基本运算，假设布尔变量A为真，变量B为假

操作符	描述	例子
&&	称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真。	（A && B）为假。
||	称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真。	（A || B）为真。
!	称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为true，则逻辑非运算符将得到false。	！（A && B）为真。

实例

下面的简单示例程序演示了逻辑运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {
  public static void main(String args[]) {
     boolean a = true;
     boolean b = false;
     System.out.println("a && b = " + (a&&b));
     System.out.println("a || b = " + (a||b) );
     System.out.println("!(a && b) = " + !(a && b));
  }
}
```

以上实例编译运行结果如下：

```
a && b = false
a || b = true
!(a && b) = true
```

赋值运算符

下面是Java语言支持的赋值运算符：

操作符	描述	例子
=	简单的赋值运算符，将右操作数的值赋给左侧操作数	C = A + B将把A + B得到的值赋给C
+=	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数	C += A等价于C = C + A
-=	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数	C -= A等价于C = C - A
*=	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数	C = *A*等价于*C* = *C* A
/=	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数	C /= A等价于C = C / A
%=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数	C %= A等价于C = C%A
<<=	左移位赋值运算符	C < = 2等价于C = C << 2
>>=	右移位赋值运算符	C >>= 2等价于C = C >> 2
&=	按位与赋值运算符	C &= 2等价于C = C&2
^=	按位异或赋值操作符	C ^= 2等价于C = C ^ 2
|=	按位或赋值操作符	C |= 2等价于C = C | 2

实例

面的简单示例程序演示了赋值运算符。复制并粘贴下面的Java程序并保存为Test.java文件，然后编译并运行这个程序：

```
public class Test {
  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 0;
     c = a + b;
     System.out.println("c = a + b = " + c );
     c += a ;
     System.out.println("c += a  = " + c );
     c -= a ;
     System.out.println("c -= a = " + c );
     c *= a ;
     System.out.println("c *= a = " + c );
     a = 10;
     c = 15;
     c /= a ;
     System.out.println("c /= a = " + c );
     a = 10;
     c = 15;
     c %= a ;
     System.out.println("c %= a  = " + c );
     c <<= 2 ;
     System.out.println("c <<= 2 = " + c );
     c >>= 2 ;
     System.out.println("c >>= 2 = " + c );
     c >>= 2 ;
     System.out.println("c >>= a = " + c );
     c &= a ;
     System.out.println("c &= 2  = " + c );
     c ^= a ;
     System.out.println("c ^= a   = " + c );
     c |= a ;
     System.out.println("c |= a   = " + c );
  }
}
```

以上实例编译运行结果如下：

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

条件运算符（?:）

条件运算符也被称为三元运算符。该运算符有3个操作数，并且需要判断布尔表达式的值。该运算符的主要是决定哪个值应该赋值给变量。

```
variable x = (expression) ? value if true : value if false
```

实例

```
public class Test {
   public static void main(String args[]){
      int a , b;
      a = 10;
b = (a == 1) ? 20: 30;
System.out.println( "Value of b is : " +  b );
      b = (a == 10) ? 20: 30;
     System.out.println( "Value of b is : " + b );
   }
}
```

以上实例编译运行结果如下：

```
Value of b is : 30
Value of b is : 20
```

instanceOf 运算符

该运算符用于操作对象实例，检查该对象是否是一个特定类型（类类型或接口类型）。

instanceof运算符使用格式如下：

```
( Object reference variable ) instanceOf  (class/interface type)
```

如果运算符左侧变量所指的对象，是操作符右侧类或接口(class/interface)的一个对象，那么结果为真。

下面是一个例子：

```
String name = 'James';
boolean result = name instanceOf String; // 由于name是Strine类型，所以返回真
```

如果被比较的对象兼容于右侧类型,该运算符仍然返回true。

看下面的例子：

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result);
   }
}
```

以上实例编译运行结果如下：

```
true
```

Java运算符优先级

当多个运算符出现在一个表达式中，谁先谁后呢？这就涉及到运算符的优先级别的问题。在一个多运算符的表达式中，运算符优先级不同会导致最后得出的结果差别甚大。

例如，（1+3）＋（3+2）*2，这个表达式如果按加号最优先计算，答案就是 18，如果按照乘号最优先，答案则是 14。

再如，x = 7 + 3 *2;这里x得到13，而不是20*，因为乘法运算符比加法运算符有较高的优先级，所以先计算*3* 2得到6，然后再加7。

下表中具有最高优先级的运算符在的表的最上面，最低优先级的在表的底部。

类别	操作符	关联性		
后缀	() [] . (点操作符)	左到右		
一元	+ + - ！?	从右到左		
乘性	* /%	左到右		
加性	+ -	左到右		
移位	>> >>> <<	左到右		
关系	>> = << =	左到右		
相等	== !=	左到右		
按位与	＆	左到右		
按位异或	^	左到右		
按位或			左到右	
逻辑与	&&	左到右		
逻辑或				左到右
条件	？：	从右到左		
赋值	= + = - = * = / =%= >> = << =&= ^ =	=	从右到左	
逗号	,	左到右		

Java循环结构 - for, while 及 do...while

顺序结构的程序语句只能被执行一次。如果您想要同样的操作执行多次,，就需要使用循环结构。

Java中有三种主要的循环结构：

- while循环
- do…while循环
- for循环

在Java5中引入了一种主要用于数组的增强型for循环。

while循环

while是最基本的循环，它的结构为：

```
while( 布尔表达式 ) {
    //循环内容
}
```

只要布尔表达式为true，循环体会一直执行下去。

实例

```
public class Test {
   public static void main(String args[]) {
      int x = 10;
      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

do…while循环

对于while语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。

do…while循环和while循环相似，不同的是，do…while循环至少会执行一次。

```
do {
       //代码语句
}while(布尔表达式);
```

注意：布尔表达式在循环体的后面，所以语句块在检测布尔表达式之前已经执行了。 如果布尔表达式的值为true，则语句块一直执行，直到布尔表达式的值为false。

实例

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      do{
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );
   }
}
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

for循环

虽然所有循环结构都可以用while或者do...while表示，但Java提供了另一种语句 —— for循环，使一些循环结构变得更加简单。

for循环执行的次数是在执行前就确定的。语法格式如下

```
for(初始化; 布尔表达式; 更新) {
    //代码语句
}
```

关于for循环有以下几点说明：

- 最先执行初始化步骤。可以声明并初始化一个或多个循环控制变量，也可以是空语句。
- 然后，检测布尔表达式的值。如果为true，循环体被执行。如果为false，循环终止，开始执行循环体后面的语句。
- 执行一次循环后，更新循环控制变量。
- 再次检测布尔表达式。循环执行上面的过程。

实例

```
public class Test {

   public static void main(String args[]) {

      for(int x = 10; x < 20; x = x+1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Java增强for循环

Java5引入了一种主要用于数组的增强型for循环。

Java增强for循环语法格式如下:

```
for(声明语句 : 表达式)
{
   //代码句子
}
```

声明语句：声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

表达式：表达式是要访问的数组名，或者是返回值为数组的方法。

实例

```
public class Test {

   public static void main(String args[]){
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

以上实例编译运行结果如下：

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

break关键字

break主要用在循环语句或者switch语句中，用来跳出整个语句块。

break跳出最里层的循环，并且继续执行该循环下面的语句。

语法

break的用法很简单，就是循环结构中的一条语句：

```
break;
```

实例

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
          break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
10
20
```

continue关键字

continue适用于任何循环控制结构中。作用是让程序立刻跳转到下一次循环的迭代。

在for循环中，continue语句使程序立即跳转到更新语句。

在while或者do…while循环中，程序立即跳转到布尔表达式的判断语句。

语法

continue就是循环体中一条简单的语句：

```
continue;
```

实例

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
          continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
10
20
40
50
```

Java分支结构 - if...else/switch

顺序结构只能顺序执行，不能进行判断和选择，因此需要分支结构。

Java有两种分支结构：

- if语句
- switch语句

if语句

一个if语句包含一个布尔表达式和一条或多条语句。

语法

If语句的用语法如下：

```
if(布尔表达式)
{
   //如果布尔表达式为true将执行的语句
}
```

如果布尔表达式的值为true，则执行if语句中的代码块。否则执行If语句块后面的代码。

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      if( x < 20 ){
         System.out.print("这是 if 语句");
      }
   }
}
```

以上代码编译运行结果如下：

```
这是 if 语句
```

if...else语句

if语句后面可以跟else语句，当if语句的布尔表达式值为false时，else语句块会被执行。

语法

if…else的用法如下：

```
if(布尔表达式){
   //如果布尔表达式的值为true
}else{
   //如果布尔表达式的值为false
}
```

实例

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x < 20 ){
         System.out.print("这是 if 语句");
      }else{
         System.out.print("这是 else 语句");
      }
   }
}
```

以上代码编译运行结果如下：

```
这是 else 语句
```

if...else if...else语句

if语句后面可以跟elseif…else语句，这种语句可以检测到多种可能的情况。

使用if，else if，else语句的时候，需要注意下面几点：

- if语句至多有1个else语句，else语句在所有的elseif语句之后。
- If语句可以有若干个elseif语句，它们必须在else语句之前。
- 一旦其中一个else if语句检测为true，其他的else if以及else语句都将跳过执行。

语法

if...else语法格式如下:

```
if(布尔表达式 1){
   //如果布尔表达式 1的值为true执行代码
}else if(布尔表达式 2){
   //如果布尔表达式 2的值为true执行代码
}else if(布尔表达式 3){
   //如果布尔表达式 3的值为true执行代码
}else {
   //如果以上布尔表达式都不为true执行代码
}
```

实例

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x == 10 ){
         System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

以上代码编译运行结果如下：

```
Value of X is 30
```

嵌套的if...else语句

使用嵌套的if-else语句是合法的。也就是说你可以在另一个if或者elseif语句中使用if或者elseif语句。

语法

嵌套的if…else语法格式如下：

```
if(布尔表达式 1){
   ////如果布尔表达式 1的值为true执行代码
   if(布尔表达式 2){
      ////如果布尔表达式 2的值为true执行代码
   }
}
```

你可以像 *if* 语句一样嵌套 *else if...else*。

实例

```
public class Test {

   public static void main(String args[]){
      int x = 30;
      int y = 10;

      if( x == 30 ){
         if( y == 10 ){
             System.out.print("X = 30 and Y = 10");
          }
       }
    }
}
```

以上代码编译运行结果如下：

```
X = 30 and Y = 10
```

switch语句

switch语句判断一个变量与一系列值中某个值是否相等，每个值称为一个分支。

语法

switch语法格式如下：

```
switch(expression){
    case value :
       //语句
       break; //可选
    case value :
       //语句
       break; //可选
    //你可以有任意数量的case语句
    default : //可选
       //语句
}
```

switch语句有如下规则：

- switch语句中的变量类型只能为byte、short、int或者char。
- switch语句可以拥有多个case语句。每个case后面跟一个要比较的值和冒号。
- case语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。
- 当变量的值与case语句的值相等时，那么case语句之后的语句开始执行，直到break语句出现才会跳出switch语句。3
- 当遇到break语句时，switch语句终止。程序跳转到switch语句后面的语句执行。case语句不必须要包含break语句。如果没有break语句出现，程序会继续执行下一条case语句，直到出现break语句。

- switch语句可以包含一个default分支，该分支必须是switch语句的最后一个分支。default在没有case语句的值和变量值相等的时候执行。default分支不需要break语句。

实例

```
public class Test {

   public static void main(String args[]){
      //char grade = args[0].charAt(0);
      char grade = 'C';

      switch(grade)
      {
         case 'A' :
            System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Well done");
            break;
         case 'D' :
            System.out.println("You passed");
         case 'F' :
            System.out.println("Better try again");
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

以上代码编译运行结果如下：

```
$ java Test
Well done
Your grade is a C
$
```

Java Number类

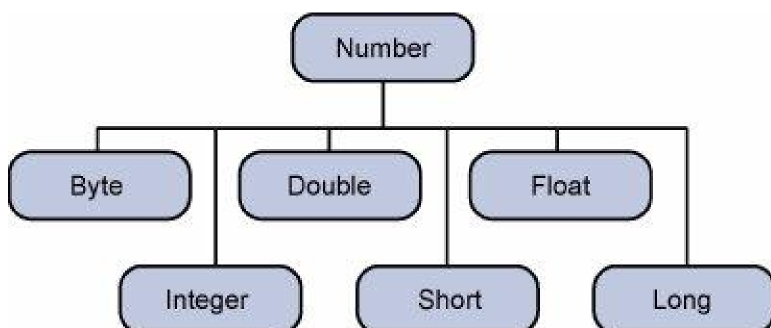
一般地，当需要使用数字的时候，我们通常使用内置数据类型，如：byte、int、long、double等。

实例

```
int i = 5000;
float gpa = 13.65;
byte mask = 0xaf;
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情形。为了解决这个问题，Java语言为每一个内置数据类型提供了对应的包装类。

所有的包装类（Integer、Long、Byte、Double、Float、Short）都是抽象类Number的子类。



这种由编译器特别支持的包装称为装箱，所以当内置数据类型被当作对象使用的时候，编译器会把内置类型装箱为包装类。相似的，编译器也可以把一个对象拆箱为内置类型。Number类属于java.lang包。

下面是一个装箱与拆箱的例子：

```
public class Test{

   public static void main(String args[]){
      Integer x = 5; // boxes int to an Integer object
      x =  x + 10;   // unboxes the Integer to a int
      System.out.println(x);
   }
}
```

以上实例编译运行结果如下：

```
15
```

当x被赋为整型值时，由于x是一个对象，所以编译器要对x进行装箱。然后，为了使x能进行加运算，所以要对x进行拆箱。

Number类的成员方法

下面的表中列出的是Number类的方法：

方法	描述
xxxValue()	将number对象转换为xxx数据类型的值并返回。
compareTo()	将number对象与参数比较。
equals()	判断number对象是否与参数相等。
valueOf()	返回一个Integer对象指定的内置数据类型
toString()	以字符串形式返回值。
parseInt()	将字符串解析为int类型。
abs()	返回参数的绝对值。
ceil()	对整形变量向左取整，返回类型为double型。
floor()	对整型变量向右取整。返回类型为double类型。
rint()	返回与参数最接近的整数。返回类型为double。
round()	返回一个最接近的int、long型值。
min()	返回两个参数中的最小值。
max()	返回两个参数中的最大值。
exp()	返回自然数底数e的参数次方。
log()	返回参数的自然数底数的对数值。
pow()	返回第一个参数的第二个参数次方。
sqrt()	求参数的算术平方根。
sin()	求指定double类型参数的正弦值。
cos()	求指定double类型参数的余弦值。
tan()	求指定double类型参数的正切值。
asin()	求指定double类型参数的反正弦值。
acos()	求指定double类型参数的反余弦值。
atan()	求指定double类型参数的反正切值。
atan2()	将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。
toDegrees()	将参数转化为角度。
toRadians()	将角度转换为弧度。
random()	返回一个随机数。

Java Character类

使用字符时，我们通常使用的是内置数据类型char。

实例

```
char ch = 'a';

// Unicode for uppercase Greek omega character
char uniChar = '\u039A';

// 字符数组
char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情况。为了解决这个问题，Java语言为内置数据类型char提供了包装类Character类。

Character类提供了一系列方法来操纵字符。你可以使用Character的构造方法创建一个Character类对象，例如：

```
Character ch = new Character('a');
```

在某些情况下，Java编译器会自动创建一个Character对象。

例如，将一个char类型的参数传递给需要一个Character类型参数的方法时，那么编译器会自动地将char类型参数转换为Character对象。 这种特征称为装箱，反过来称为拆箱。

实例

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch = 'a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

转义序列

前面有反斜杠（\）的字符代表转义字符，它对编译器来说是有特殊含义的。

下面列表展示了Java的转义序列：

转义序列	描述
\t	在文中该处插入一个tab键
\b	在文中该处插入一个后退键
\n	在文中该处换行
\r	在文中该处插入回车
\f	在文中该处插入换页符
\'	在文中该处插入单引号
\"	在文中该处插入双引号
\	在文中该处插入反斜杠

实例

当打印语句遇到一个转义序列时，编译器可以正确地对其进行解释。

```
public class Test {

   public static void main(String args[]) {
      System.out.println("She said \"Hello!\" to me.");
   }
}
```

以上实例编译运行结果如下：

```
She said "Hello!" to me.
```

Character 方法

下面是Character类的方法：

方法	描述
isLetter()	是否是一个字母
isDigit()	是否是一个数字字符
isWhitespace()	是否一个空格
isUpperCase()	是否是大写字母
isLowerCase()	是否是小写字母
toUpperCase()	指定字母的大写形式
toLowerCase()	指定字母的小写形式
toString()	返回字符的字符串形式，字符串的长度仅为1

对于方法的完整列表，请参考的java.lang.Character API规范。

Java String类

字符串广泛应用在Java编程中，在Java中字符串属于对象，Java提供了String类来创建和操作字符串。

创建字符串

创建字符串最简单的方式如下:

```
String greeting = "Hello world!";
```

在代码中遇到字符串常量时，这里的值是"Hello world!"，编译器会使用该值创建一个String对象。

和其它对象一样，可以使用关键字和构造方法来创建String对象。

String类有11种构造方法，这些方法提供不同的参数来初始化字符串，比如提供一个字符数组参数:

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

以上实例编译运行结果如下：

```
hello.
```

注意:String类是不可改变的，所以你一旦创建了String对象，那它的值就无法改变了。 如果需要对字符串做很多修改，那么应该选择使用StringBuffer & StringBuilder 类。

字符串长度

用于获取有关对象的信息的方法称为访问器方法。

String类的一个访问器方法是length()方法，它返回字符串对象包含的字符数。

下面的代码执行后，len变量等于17:

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
   }
}
```

以上实例编译运行结果如下：

```
String Length is : 17
```

连接字符串

String类提供了连接两个字符串的方法：

```
string1.concat(string2);
```

返回string2连接string1的新字符串。也可以对字符串常量使用concat()方法，如：

```
"My name is ".concat("Zara");
```

更常用的是使用'+'操作符来连接字符串，如：

```
"Hello," + " world" + "!"
```

结果如下:

```
"Hello, world!"
```

下面是一个例子:

```
public class StringDemo {
   public static void main(String args[]) {
   String string1 = "saw I was ";
   System.out.println("Dot " + string1 + "Tod");
}
}
```

以上实例编译运行结果如下：

```
Dot saw I was Tod
```

创建格式化字符串

我们知道输出格式化数字可以使用printf()和format()方法。String类使用静态方法format()返回一个String对象而不是PrintStream对象。

String类的静态方法format()能用来创建可复用的格式化字符串，而不仅仅是用于一次打印输出。如下所示：

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

你也可以这样写

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String 方法

下面是String类支持的方法，更多详细，参看Java API文档:

方法	描述
char charAt(int index)	返回指定索引处的 char 值。
int compareTo(Object o)	把这个字符串和另一个对象比较。
int compareTo(String anotherString)	按字典顺序比较两个字符串。
int compareToIgnoreCase(String str)	按字典顺序比较两个字符串，不考虑大小写。
String concat(String str)	将指定字符串连接到此字符串的结尾。
boolean contentEquals(StringBuffer sb)	当且仅当字符串与指定的StringButter有相同顺序的字符时候返回真。
static String copyValueOf(char[] data)	返回指定数组中表示该字符序列的 String。
static String copyValueOf(char[] data, int offset, int count)	返回指定数组中表示该字符序列的 String。
boolean endsWith(String suffix)	测试此字符串是否以指定的后缀结束。
boolean equals(Object anObject)	将此字符串与指定的对象比较。
boolean equalsIgnoreCase(String anotherString)	将此 String 与另一个 String 比较，不考虑大小写。

byte[] getBytes()	使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
byte[] getBytes(String charsetName)	使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	将字符从此字符串复制到目标字符数组。
int hashCode()	返回此字符串的哈希码。
int indexOf(int ch)	返回指定字符在此字符串中第一次出现处的索引。
int indexOf(int ch, int fromIndex)	返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
int indexOf(String str)	返回指定子字符串在此字符串中第一次出现处的索引。
int indexOf(String str, int fromIndex)	返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。
String intern()	返回字符串对象的规范化表示形式。
int lastIndexOf(int ch)	返回指定字符在此字符串中最后一次出现处的索引。
int lastIndexOf(int ch, int fromIndex)	返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。
int lastIndexOf(String str)	返回指定子字符串在此字符串中最右边出现处的索引。
int lastIndexOf(String str, int fromIndex)	返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。
int length()	返回此字符串的长度。
boolean matches(String regex)	告知此字符串是否匹配给定的正则表达式。
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等。
boolean regionMatches(int toffset, String other, int ooffset, int len)	测试两个字符串区域是否相等。
String replace(char oldChar, char newChar)	返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
String replaceAll(String regex, String replacement)	使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
String replaceFirst(String regex, String	使用给定的 replacement 替换此字符串匹配

replacement)	给定的正则表达式的第一个子字符串。
String[] split(String regex)	根据给定正则表达式的匹配拆分此字符串。
String[] split(String regex, int limit)	根据匹配给定的正则表达式来拆分此字符串。
boolean startsWith(String prefix)	测试此字符串是否以指定的前缀开始。
boolean startsWith(String prefix, int toffset)	测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
CharSequence subSequence(int beginIndex, int endIndex)	返回一个新的字符序列，它是此序列的一个子序列。
String substring(int beginIndex)	返回一个新的字符串，它是此字符串的一个子字符串。
String substring(int beginIndex, int endIndex)	返回一个新字符串，它是此字符串的一个子字符串。
char[] toCharArray()	将此字符串转换为一个新的字符数组。
String toLowerCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为小写。
String toLowerCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。
String toString()	返回此对象本身（它已经是一个字符串！）。
String toUpperCase()	使用默认语言环境的规则将此 String 中的所有字符都转换为大写。
String toUpperCase(Locale locale)	使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。
String trim()	返回字符串的副本，忽略前导空白和尾部空白。
static String valueOf(primitive data type x)	返回给定data type类型x参数的字符串表示形式。

Java StringBuffer和StringBuilder类

当对字符串进行修改的时候，需要使用StringBuffer和StringBuilder类。

和String类不同的是，StringBuffer和StringBuilder类的对象能够被多次的修改，并且不产生新的未使用对象。

StringBuilder类在Java 5中被提出，它和StringBuffer之间的最大不同在于StringBuilder的方法不是线程安全的（不能同步访问）。

由于StringBuilder相较于StringBuffer有速度优势，所以多数情况下建议使用StringBuilder类。然而在应用程序要求线程安全的情况下，则必须使用StringBuffer类。

实例

```
public class Test{

    public static void main(String args[]){
       StringBuffer sBuffer = new StringBuffer(" test");
       sBuffer.append(" String Buffer");
       System.ou.println(sBuffer);
   }
}
```

以上实例编译运行结果如下：

```
test String Buffer
```

StringBuffer 方法

以下是StringBuffer类支持的主要方法：

方法	描述
public StringBuffer append(String s)	将指定的字符串追加到此字符序列。
public StringBuffer reverse()	将此字符序列用其反转形式取代。
public delete(int start, int end)	移除此序列的子字符串中的字符。
public insert(int offset, int i)	将 `int` 参数的字符串表示形式插入此序列中。
replace(int start, int end, String str)	使用给定 `String` 中的字符替换此序列的子字符串中的字符。

下面的列表里的方法和String类的方法类似：

方法	描述
int capacity()	返回当前容量。
char charAt(int index)	返回此序列中指定索引处的 `char` 值。
void ensureCapacity(int minimumCapacity)	确保容量至少等于指定的最小值。
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	将字符从此序列复制到目标字符数组 `dst` 。
int indexOf(String str)	返回第一次出现的指定子字符串在该字符串中的索引。
int indexOf(String str, int fromIndex)	从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。
int lastIndexOf(String str)	返回最右边出现的指定子字符串在此字符串中的索引。
int lastIndexOf(String str, int fromIndex)	返回最后一次出现的指定子字符串在此字符串中的索引。
int length()	返回长度（字符数）。
void setCharAt(int index, char ch)	将给定索引处的字符设置为 `ch` 。
void setLength(int newLength)	设置字符序列的长度。
CharSequence subSequence(int start, int end)	返回一个新的字符序列，该字符序列是此序列的子序列。
String substring(int start)	返回一个新的 `String` ，它包含此字符序列当前所包含的字符子序列。
String substring(int start, int end)	返回一个新的 `String` ，它包含此序列当前所包含的字符子序列。
String toString()	返回此序列中数据的字符串表示形式。

Java 数组

数组对于每一门编辑应语言来说都是重要的数据结构之一，当然不同语言对数组的实现及处理也不尽相同。

Java语言中提供的数组是用来存储固定大小的同类型元素。

你可以声明一个数组变量，如numbers[100]来代替直接声明100个独立变量number0，number1，....，number99。

本教程将为大家介绍Java数组的声明、创建和初始化，并给出其对应的代码。

声明数组变量

首先必须声明数组变量，才能在程序中使用数组。下面是声明数组变量的语法：

```
dataType[] arrayRefVar;   // 首选的方法

或

dataType arrayRefVar[];  // 效果相同，但不是首选方法
```

注意: 建议使用dataType[] arrayRefVar 的声明风格声明数组变量。 dataType arrayRefVar[] 风格是来自 C/C++ 语言 ， 在Java中采用是为了让 C/C++ 程序员能够快速理解java语言。

实例

下面是这两种语法的代码示例：

```
double[] myList;         // 首选的方法

或

double myList[];         //  效果相同，但不是首选方法
```

创建数组

Java语言使用new操作符来创建数组，语法如下：

```
arrayRefVar = new dataType[arraySize];
```

上面的语法语句做了两件事：

- 一、使用dataType[arraySize]创建了一个数组。
- 二、把新创建的数组的引用赋值给变量 arrayRefVar。

数组变量的声明，和创建数组可以用一条语句完成，如下所示：

```
dataType[] arrayRefVar = new dataType[arraySize];
```

另外，你还可以使用如下的方式创建数组。

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

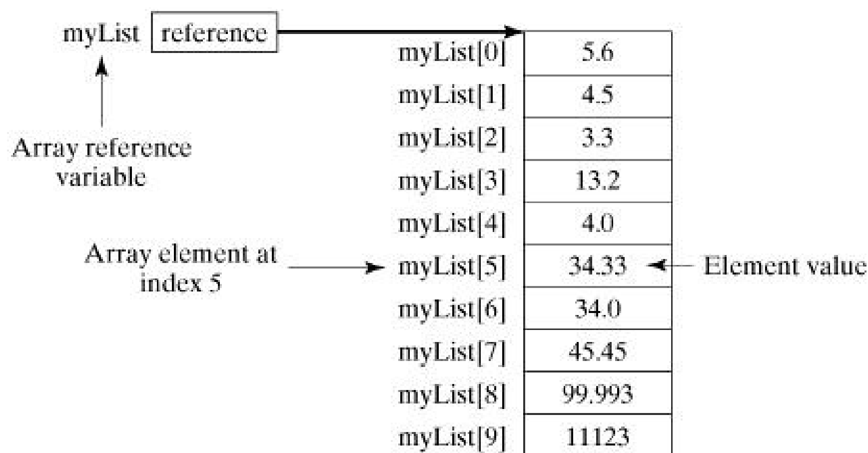
数组的元素是通过索引访问的。数组索引从0开始，所以索引值从0到arrayRefVar.length-1。

实例

下面的语句首先声明了一个数组变量myList，接着创建了一个包含10个double类型元素的数组，并且把它的引用赋值给myList变量。

```
double[] myList = new double[10];
```

下面的图片描绘了数组myList。这里myList数组里有10个double元素，它的下标从0到9。



处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本循环或者foreach循环。

示例

该实例完整地展示了如何创建、初始化和操纵数组：

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // 打印所有数组元素
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }
      // 计算所有元素的总和
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);
      // 查找最大元素
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

以上实例编译运行结果如下：

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

foreach循环

JDK 1.5 引进了一种新的循环类型，被称为foreach循环或者加强型循环，它能在不使用下标的情况下遍历数组。

示例

该实例用来显示数组myList中的所有元素：

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // 打印所有数组元素
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

以上实例编译运行结果如下：

```
1.9
2.9
3.4
3.5
```

数组作为函数的参数

数组可以作为参数传递给方法。例如，下面的例子就是一个打印int数组中元素的方法。

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

下面例子调用printArray方法打印出 3，1，2，6，4和2：

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

数组作为函数的返回值

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
}
```

以上实例中result数组作为函数的返回值。

Arrays 类

java.util.Arrays类能方便地操作数组，它提供的所有方法都是静态的。具有以下功能：

- 给数组赋值：通过fill方法。
- 对数组排序：通过sort方法,按升序。
- 比较数组：通过equals方法比较数组中元素值是否相等。
- 查找数组元素：通过binarySearch方法能对排序好的数组进行二分查找法操作。

具体说明请查看下表：

方法	说明
public static int binarySearch(Object[] a, Object key)	用二分查找算法在给定数组中搜索给定值的对象(Byte,Int,double等)。数组在调用前必须排序好的。如果查找值包含在数组中，则返回搜索键的索引；否则返回 (-(插入点) - 1)。
public static boolean equals(long[] a, long[] a2)	如果两个指定的 long 型数组彼此相等，则返回 true。如果两个数组包含相同数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。换句话说，如果两个数组以相同顺序包含相同的元素，则两个数组是相等的。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。
public static void fill(int[] a, int val)	将指定的 int 值分配给指定 int 型数组指定范围中的每个元素。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。
public static void sort(Object[] a)	对指定对象数组根据其元素的自然顺序进行升序排列。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。

Java 日期时间

java.util包提供了Date类来封装当前的日期和时间。 Date类提供两个构造函数来实例化Date对象。

第一个构造函数使用当前日期和时间来初始化对象。

```
Date( )
```

第二个构造函数接收一个参数，该参数是从1970年1月1日起的微秒数。

```
Date(long millisec)
```

Date对象创建以后，可以调用下面的方法。

方法	描述
boolean after(Date date)	若当调用此方法的Date对象在指定日期之后返回true,否则返回false。
boolean before(Date date)	若当调用此方法的Date对象在指定日期之前返回true,否则返回false。
Object clone( )	返回此对象的副本。
int compareTo(Date date)	比较当调用此方法的Date对象和指定日期。两者相等时候返回0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。
int compareTo(Object obj)	若obj是Date类型则操作等同于compareTo(Date) 。否则它抛出ClassCastException。
boolean equals(Object date)	当调用此方法的Date对象和指定日期相等时候返回true,否则返回false。
long getTime( )	返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
int hashCode( )	返回此对象的哈希码值。
void setTime(long time)	用自1970年1月1日00:00:00 GMT以后time毫秒数设置时间和日期。
String toString( )	转换Date对象为String表示形式，并返回该字符串。

获取当前日期时间

Java中获取当前日期和时间很简单，使用Date对象的 toString()方法来打印当前日期和时间，如下所示：

```
import java.util.Date;

public class DateDemo {
   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 使用 toString() 函数显示日期时间
       System.out.println(date.toString());
   }
}
```

以上实例编译运行结果如下:

```
Mon May 04 09:51:52 CDT 2013
```

日期比较

Java使用以下三种方法来比较两个日期：

- 使用getTime( ) 方法获取两个日期（自1970年1月1日经历的微妙数值），然后比较这两个值。
- 使用方法before()，after()和equals()。例如，一个月的12号比18号早，则new Date(99, 2, 12).before(new Date (99, 2, 18))返回true。
- 使用compareTo()方法，它是由Comparable接口定义的，Date类实现了这个接口。

使用**SimpleDateFormat**格式化日期

SimpleDateFormat是一个以语言环境敏感的方式来格式化和分析日期的类。SimpleDateFormat允许你选择任何用户自定义日期时间格式来运行。例如：

```
import java.util.*;
import java.text.*;

public class DateDemo {
   public static void main(String args[]) {

      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

      System.out.println("Current Date: " + ft.format(dNow));
   }
}
```

以上实例编译运行结果如下:

```
Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
```

简单的DateFormat格式化编码

时间模式字符串用来指定时间格式。在此模式中，所有的ASCII字母被保留为模式字母，定义如下：

字母	描述	示例
G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07
d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	微妙数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40
W	一个月中第几周	1
a	A.M./P.M. 标记	PM
k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	`

使用printf格式化日期

printf方法可以很轻松地格式化时间和日期。使用两个字母格式，它以t开头并且以下面表格中的一个字母结尾。例如：

```
import java.util.Date;

public class DateDemo {

  public static void main(String args[]) {
     // 初始化 Date 对象
     Date date = new Date();

     // 使用toString()显示日期和时间
     String str = String.format("Current Date/Time : %tc", date );

     System.out.printf(str);
  }
}
```

以上实例编译运行结果如下:

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

如果你需要重复提供日期，那么利用这种方式来格式化它的每一部分就有点复杂了。因此，可以利用一个格式化字符串指出要被格式化的参数的索引。

索引必须紧跟在%后面，而且必须以$结束。例如：

```
import java.util.Date;

public class DateDemo {

   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 使用toString()显示日期和时间
       System.out.printf("%1$s %2$tB %2$td, %2$tY",
                         "Due date:", date);
   }
}
```

以上实例编译运行结果如下:

```
Due date: February 09, 2004
```

或者，你可以使用<标志。它表明先前被格式化的参数要被再次使用。例如：

```
import java.util.Date;

public class DateDemo {

   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 显示格式化时间
       System.out.printf("%s %tB %<te, %<tY",
                         "Due date:", date);
   }
}
```

以上实例编译运行结果如下:

```
Due date: February 09, 2004
```

日期和时间转换字符

字符	描述	例子
c	完整的日期和时间	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 格式日期	2004-02-09
D	U.S. 格式日期 (月/日/年)	02/09/2004
T	24小时时间	18:05:19
r	12小时时间	06:05:19 pm
R	24小时时间，不包含秒	18:05
Y	4位年份(包含前导0)	2004
y	年份后2位(包含前导0)	04
C	年份前2位(包含前导0)	20
B	月份全称	February
b	月份简称	Feb
n	2位月份(包含前导0)	02
d	2位日子(包含前导0)	03
e	2位日子(不包含前导0)	9
A	星期全称	Monday
a	星期简称	Mon
j	3位年份(包含前导0)	069
H	2位小时(包含前导0), 00 到 23	18
k	2位小时(不包含前导0), 0 到 23	18
I	2位小时(包含前导0), 01 到 12	06
l	2位小时(不包含前导0), 1 到 12	6
M	2位分钟(包含前导0)	05
S	2位秒数(包含前导0)	19
L	3位毫秒(包含前导0)	047
N	9位纳秒(包含前导0)	047000000

P	大写上下午标志	PM
p	小写上下午标志	pm
z	从GMT的RFC 822数字偏移	-0800
Z	时区	PST
s	自 1970-01-01 00:00:00 GMT的秒数	1078884319
Q	自 1970-01-01 00:00:00 GMT的毫妙	1078884319047

还有其他有用的日期和时间相关的类。对于更多的细节，你可以参考到Java标准文档。

解析字符串为时间

SimpleDateFormat 类有一些附加的方法，特别是parse()，它试图按照给定的SimpleDateFormat 对象的格式化存储来解析字符串。例如：

```
import java.util.*;
import java.text.*;

public class DateDemo {

   public static void main(String args[]) {
      SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

      String input = args.length == 0 ? "1818-11-11" : args[0];

      System.out.print(input + " Parses as ");

      Date t;

      try {
          t = ft.parse(input);
          System.out.println(t);
      } catch (ParseException e) {
          System.out.println("Unparseable using " + ft);
      }
   }
}
```

以上实例编译运行结果如下:

```
$ java DateDemo
1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818
$ java DateDemo 2007-12-01
2007-12-01 Parses as Sat Dec 01 00:00:00 GMT 2007
```

Java 休眠(sleep)

你可以让程序休眠一毫秒的时间或者到您的计算机的寿命长的任意段时间。例如，下面的程序会休眠10秒：

```
import java.util.*;

public class SleepDemo {
   public static void main(String args[]) {
      try {
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
      } catch (Exception e) {
          System.out.println("Got an exception!");
      }
   }
}
```

以上实例编译运行结果如下:

```
Sun May 03 18:04:41 GMT 2009

Sun May 03 18:04:51 GMT 2009
```

测量时间

下面的一个例子表明如何测量时间间隔（以毫秒为单位）：

```
import java.util.*;

public class DiffDemo {

   public static void main(String args[]) {
      try {
         long start = System.currentTimeMillis( );
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
         long end = System.currentTimeMillis( );
         long diff = end - start;
         System.out.println("Difference is : " + diff);
      } catch (Exception e) {
         System.out.println("Got an exception!");
      }
   }
}
```

以上实例编译运行结果如下:

```
Sun May 03 18:16:51 GMT 2009

Sun May 03 18:16:57 GMT 2009

Difference is : 5993
```

Calendar类

我们现在已经能够格式化并创建一个日期对象了，但是我们如何才能设置和获取日期数据的特定部分呢，比如说小时，日，或者分钟？我们又如何在日期的这些部分加上或者减去值呢？答案是使用Calendar 类。

Calendar类的功能要比Date类强大很多，而且在实现方式上也比Date类要复杂一些。

Calendar类是一个抽象类，在实际使用时实现特定的子类的对象，创建对象的过程对程序员来说是透明的，只需要使用getInstance方法创建即可。

创建一个代表系统当前日期的Calendar对象

```
Calendar c = Calendar.getInstance();//默认是当前日期
```

创建一个指定日期的Calendar对象

使用Calendar类代表特定的时间，需要首先创建一个Calendar的对象，然后再设定该对象中的年月日参数来完成。

```
//创建一个代表2009年6月12日的Calendar对象
Calendar c1 = Calendar.getInstance();
c1.set(2009, 6 - 1, 12);
```

Calendar类对象字段类型

Calendar类中用一下这些常量表示不同的意义，jdk内的很多类其实都是采用的这种思想

常量	描述
Calendar.YEAR	年份
Calendar.MONTH	月份
Calendar.DATE	日期
Calendar.DAY_OF_MONTH	日期，和上面的字段意义完全相同
Calendar.HOUR	12小时制的小时
Calendar.HOUR_OF_DAY	24小时制的小时
Calendar.MINUTE	分钟
Calendar.SECOND	秒
Calendar.DAY_OF_WEEK	星期几

Calendar类对象信息的设置

Set设置

如：

```
Calendar c1 = Calendar.getInstance();
```

调用：

```
public final void set(int year,int month,int date)
```

```
c1.set(2009, 6 - 1, 12);//把Calendar对象c1的年月日分别设这为：2009、6、12
```

利用字段类型设置

如果只设定某个字段，例如日期的值，则可以使用如下set方法：

```
public void set(int field,int value)
```

把 c1对象代表的日期设置为10号，其它所有的数值会被重新计算

```
c1.set(Calendar.DATE,10);
```

把c1对象代表的年份设置为2008年，其他的所有数值会被重新计算

```
c1.set(Calendar.YEAR,2008);
```

其他字段属性set的意义以此类推

Add设置

```
Calendar c1 = Calendar.getInstance();
```

把c1对象的日期加上10，也就是c1所表的日期的10天后的日期，其它所有的数值会被重新计算

```
c1.add(Calendar.DATE, 10);
```

把c1对象的日期加上10，也就是c1所表的日期的10天前的日期，其它所有的数值会被重新计算

```
<pre>c1.add(Calendar.DATE, -10);
```

其他字段属性的add的意义以此类推

Calendar类对象信息的获得

```
Calendar c1 = Calendar.getInstance();
// 获得年份
int year = c1.get(Calendar.YEAR);
// 获得月份
int month = c1.get(Calendar.MONTH) + 1;
// 获得日期
int date = c1.get(Calendar.DATE);
// 获得小时
int hour = c1.get(Calendar.HOUR_OF_DAY);
// 获得分钟
int minute = c1.get(Calendar.MINUTE);
// 获得秒
int second = c1.get(Calendar.SECOND);
// 获得星期几（注意（这个与Date类是不同的）：1代表星期日、2代表星期1、3代表星期二，以此类推）
int day = c1.get(Calendar.DAY_OF_WEEK);
```

GregorianCalendar类

Calendar类实现了公历日历，GregorianCalendar是Calendar类的一个具体实现。

Calendar 的getInstance（）方法返回一个默认用当前的语言环境和时区初始化的GregorianCalendar对象。GregorianCalendar定义了两个字段：AD和BC。这些代表公历定义的两个时代。

下面列出GregorianCalendar对象的几个构造方法：

构造函数	说明
GregorianCalendar()	在具有默认语言环境的默认时区内使用当前时间构造一个默认的 GregorianCalendar。
GregorianCalendar(int year, int month, int date)	在具有默认语言环境的默认时区内构造一个带有给定日期设置的 GregorianCalendar
GregorianCalendar(int year, int month, int date, int hour, int minute)	为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
GregorianCalendar(int year, int month, int date, int hour, int minute, int second)	为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
GregorianCalendar(Locale aLocale)	在具有给定语言环境的默认时区内构造一个基于当前时间的 GregorianCalendar。
GregorianCalendar(TimeZone zone)	在具有默认语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。
GregorianCalendar(TimeZone zone, Locale aLocale)	在具有给定语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。

这里是GregorianCalendar 类提供的一些有用的方法列表：

方法	说明
void add(int field, int amount)	根据日历规则，将指定的（有符号的）时间量添加到给定的日历字段中。
protected void computeFields()	转换UTC毫秒值为时间域值
protected void computeTime()	覆盖Calendar ，转换时间域值为UTC毫秒值
boolean equals(Object obj)	比较此 GregorianCalendar 与指定的 Object。
int get(int field)	获取指定字段的时间值
int getActualMaximum(int field)	返回当前日期，给定字段的最大值
int getActualMinimum(int field)	返回当前日期，给定字段的最小值
int getGreatestMinimum(int field)	返回此 GregorianCalendar 实例给定日历字段的最高的最小值。
Date getGregorianChange()	获得格里高利历的更改日期。
int getLeastMaximum(int field)	返回此 GregorianCalendar 实例给定日历字段的最低的最大值
int getMaximum(int field)	返回此 GregorianCalendar 实例的给定日历字段的最大值。
Date getTime()	获取日历当前时间。
long getTimeInMillis()	获取用长整型表示的日历的当前时间
TimeZone getTimeZone()	获取时区。
int getMinimum(int field)	返回给定字段的最小值。
int hashCode()	重写hashCode.
boolean isLeapYear(int year)	确定给定的年份是否为闰年。
void roll(int field, boolean up)	在给定的时间字段上添加或减去（上/下）单个时间单元，不更改更大的字段。
void set(int field, int value)	用给定的值设置时间字段。
void set(int year, int month, int date)	设置年、月、日的值。
void set(int year, int month, int date, int hour, int minute)	设置年、月、日、小时、分钟的值。
void set(int year, int month, int date, int hour, int minute, int second)	设置年、月、日、小时、分钟、秒的值。
void setGregorianChange(Date date)	设置 GregorianCalendar 的更改日期。
void setTime(Date date)	用给定的日期设置Calendar的当前时间。

void setTimeInMillis(long millis)	用给定的long型毫秒数设置Calendar的当前时间。
void setTimeZone(TimeZone value)	用给定时区值设置当前时区。
String toString()	返回代表日历的字符串。

实例

```
import java.util.*;

public class GregorianCalendarDemo {

   public static void main(String args[]) {
      String months[] = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};

      int year;
      // 初始化 Gregorian 日历
      // 使用当前时间和日期
      // 默认为本地时间和时区
      GregorianCalendar gcalendar = new GregorianCalendar();
      // 显示当前时间和日期的信息
      System.out.print("Date: ");
      System.out.print(months[gcalendar.get(Calendar.MONTH)]);
      System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
      System.out.println(year = gcalendar.get(Calendar.YEAR));
      System.out.print("Time: ");
      System.out.print(gcalendar.get(Calendar.HOUR) + ":");
      System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
      System.out.println(gcalendar.get(Calendar.SECOND));

      // 测试当前年份是否为闰年
      if(gcalendar.isLeapYear(year)) {
         System.out.println("当前年份是闰年");
      }
      else {
         System.out.println("当前年份不是闰年");
      }
   }
}
```

以上实例编译运行结果如下：

```
Date: Apr 22 2009
Time: 11:25:27
当前年份不是闰年
```

关于Calender 类的完整列表，你可以参考标准的Java文档。

Java正则表达式

正则表达式定义了字符串的模式。

正则表达式可以用来搜索、编辑或处理文本。

正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。

Java正则表达式和Perl的是最为相似的。

java.util.regex包主要包括以下三个类：

- **Pattern**类：

  pattern对象是一个正则表达式的编译表示。Pattern类没有公共构造方法。要创建一个Pattern对象，你必须首先调用其公共静态编译方法，它返回一个Pattern对象。该方法接受一个正则表达式作为它的第一个参数。

- **Matcher**类：

  Matcher对象是对输入字符串进行解释和匹配操作的引擎。与Pattern类一样，Matcher也没有公共构造方法。你需要调用Pattern对象的matcher方法来获得一个Matcher对象。

- **PatternSyntaxException**：

  PatternSyntaxException是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

捕获组

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式(dog) 创建了单一分组，组里包含"d"，"o"，和"g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式（（A）（B（C）））,有四个这样的组：

- ((A)(B(C)))
- (A)
- (B(C))
- (C)

可以通过调用matcher对象的groupCount方法来查看表达式有多少个分组。groupCount方法返回一个int值，表示matcher对象当前有多个捕获组。

还有一个特殊的组（组0），它总是代表整个表达式。该组不包括在groupCount的返回值中。

实例

下面的例子说明如何从一个给定的字符串中找到数字串：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    public static void main( String args[] ){

      // 按指定模式在字符串查找
      String line = "This order was placed for QT3000! OK?";
      String pattern = "(.*)(\\d+)(.*)";

      // 创建 Pattern 对象
      Pattern r = Pattern.compile(pattern);

      // 现在创建 matcher 对象
      Matcher m = r.matcher(line);
      if (m.find( )) {
         System.out.println("Found value: " + m.group(0) );
         System.out.println("Found value: " + m.group(1) );
         System.out.println("Found value: " + m.group(2) );
      } else {
         System.out.println("NO MATCH");
      }
   }
}
```

以上实例编译运行结果如下：

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

正则表达式语法

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\\"匹配"\"，"\("匹配"("。
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配"z"和"zoo"。 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，

+	但与"z"不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{*n*,*m*}	*M* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（*、+、?、{*n*}、{*n*,}、{*n*,*m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
.	匹配除"\n"之外的任何单个字符。若要匹配包括"\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"("或者")"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。
(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的"Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	匹配 *x* 或 *y*。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"abc"匹配"plain"中的"p"。
[*a-z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a-z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。

\b	的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\cx	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 0-9。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"[^A-Za-z0-9_]"等效。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
_n_	标识一个八进制转义码或反向引用。如果 _n 前面至少有 _*n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
_nm_	标识一个八进制转义码或反向引用。如果 _nm 前面至少有 _*nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 _nm 前面至少有 _*n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 _nm 匹配八进制值 _*nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。
\nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。
\u*n*	匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (?)。

Mather类的方法

索引方法

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

方法	说明
public int start()	返回以前匹配的初始索引。
public int start(int group)	返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
public int end()	返回最后匹配字符之后的偏移量。
public int end(int group)	返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

研究方法

研究方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

方法	说明	
public boolean lookingAt()	尝试将从区域开头开始的输入序列与该模式匹配。	
public boolean find()	尝试查找与该模式匹配的输入序列的下一个子序列。	
public boolean find(int start	)	重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
public boolean matches()	尝试将整个区域与模式匹配。	

替换方法

替换方法是替换输入字符串里文本的方法：

方法	说明
public Matcher appendReplacement(StringBuffer sb, String replacement)	实现非终端添加和替换步骤。
public StringBuffer appendTail(StringBuffer sb)	实现终端添加和替换步骤。
public String replaceAll(String replacement)	替换模式与给定替换字符串相匹配的输入序列的每个子序列。
public String replaceFirst(String replacement)	替换模式与给定替换字符串匹配的输入序列的第一个子序列。
public static String quoteReplacement(String s)	返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。

start 和end 方法

下面是一个对单词"cat"出现在输入字符串中出现次数进行计数的例子：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT =
                                    "cat cat cat cattie cat";

    public static void main( String args[] ){
       Pattern p = Pattern.compile(REGEX);
       Matcher m = p.matcher(INPUT); // 获取 matcher 对象
       int count = 0;

       while(m.find()) {
         count++;
         System.out.println("Match number "+count);
         System.out.println("start(): "+m.start());
         System.out.println("end(): "+m.end());
      }
   }
}
```

以上实例编译运行结果如下：

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。

Start方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end方法最后一个匹配字符的索引加1。

matches 和lookingAt 方法

matches 和lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是matcher要求整个序列都匹配，而lookingAt 不要求。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main( String args[] ){
       pattern = Pattern.compile(REGEX);
       matcher = pattern.matcher(INPUT);

       System.out.println("Current REGEX is: "+REGEX);
       System.out.println("Current INPUT is: "+INPUT);

       System.out.println("lookingAt(): "+matcher.lookingAt());
       System.out.println("matches(): "+matcher.matches());
   }
}
```

以上实例编译运行结果如下：

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

replaceFirst 和replaceAll 方法

replaceFirst 和replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " +
                                    "All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
       Pattern p = Pattern.compile(REGEX);
       // get a matcher object
       Matcher m = p.matcher(INPUT);
       INPUT = m.replaceAll(REPLACE);
       System.out.println(INPUT);
   }
}
```

以上实例编译运行结果如下：

```
The cat says meow. All cats say meow.
```

appendReplacement 和 appendTail 方法

Matcher 类也提供了appendReplacement 和appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
   private static String REGEX = "a*b";
   private static String INPUT = "aabfooaabfooabfoob";
   private static String REPLACE = "-";
   public static void main(String[] args) {
      Pattern p = Pattern.compile(REGEX);
      // 获取 matcher 对象
      Matcher m = p.matcher(INPUT);
      StringBuffer sb = new StringBuffer();
      while(m.find()){
         m.appendReplacement(sb,REPLACE);
      }
      m.appendTail(sb);
      System.out.println(sb.toString());
   }
}
```

以上实例编译运行结果如下：

```
-foo-foo-foo-
```

PatternSyntaxException 类的方法

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误。

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

方法	说明
public String getDescription()	获取错误的描述。
public int getIndex()	获取错误的索引。
public String getPattern()	获取错误的正则表达式模式。
public String getMessage()	返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。

Java 方法

在前面几个章节中我们经常使用到System.out.println()，那么它是什么呢？

println()是一个方法(Method)，而System是系统类(Class)，out是标准输出对象(Object)。这句话的用法是调用系统类System中的标准输出对象out中的方法println()。

那么什么是方法呢？

Java方法是语句的集合，它们在一起执行一个功能。

- 方法是解决一类问题的步骤的有序组合
- 方法包含于类或对象中
- 方法在程序中被创建，在其他地方被引用

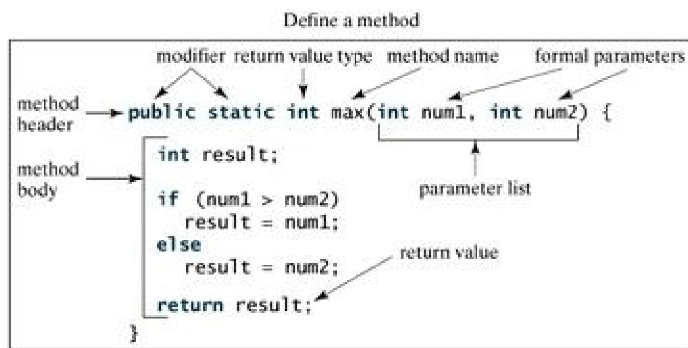
方法的定义

一般情况下，定义一个方法包含以下语法：

```
修饰符 返回值类型 方法名 (参数类型 参数名){
    ...
    方法体
    ...
    return 返回值;
}
```

方法包含一个方法头和一个方法体。下面是一个方法的所有部分：

- 修饰符：修饰符，这是可选的，告诉编译器如何调用该方法。定义了该方法的访问类型。
- 返回值类型 ：方法可能会返回值。returnValueType是方法返回值的数据类型。有些方法执行所需的操作，但没有返回值。在这种情况下，returnValueType是关键字**void**。
- 方法名：是方法的实际名称。方法名和参数表共同构成方法签名。
- 参数类型：参数像是一个占位符。当方法被调用时，传递值给参数。这个值被称为实参或变量。参数列表是指方法的参数类型、顺序和参数的个数。参数是可选的，方法可以不包含任何参数。
- 方法体：方法体包含具体的语句，定义该方法的功能。



如：

```
public static int age(int birthday){...}
```

参数可以有多个：

```
static float interest(float principal, int year){...}
```

注意： 在一些其它语言中方法指过程和函数。一个返回非void类型返回值的方法称为函数；一个返回void类型返回值的方法叫做过程。

实例

下面的方法包含2个参数num1和num2，它返回这两个参数的最大值。

```
/** 返回两个整型变量数据的较大值 */
public static int max(int num1, int num2) {
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

方法调用

Java支持两种调用方法的方式，根据方法是否返回值来选择。

当程序调用一个方法时，程序的控制权交给了被调用的方法。当被调用方法的返回语句执行或者到达方法体闭括号时候交还控制权给程序。

当方法返回一个值的时候，方法调用通常被当做一个值。例如：

```
int larger = max(30, 40);
```

如果方法返回值是void，方法调用一定是一条语句。例如，方法println返回void。下面的调用是个语句：

```
System.out.println("Welcome to Java!");
```

示例

下面的例子演示了如何定义一个方法，以及如何调用它：

```
public class TestMax {
   /** 主方法 */
   public static void main(String[] args) {
      int i = 5;
      int j = 2;
      int k = max(i, j);
      System.out.println("The maximum between " + i +
                    " and " + j + " is " + k);
   }

   /** 返回两个整数变量较大的值 */
   public static int max(int num1, int num2) {
      int result;
      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
}
```

以上实例编译运行结果如下：

```
The maximum between 5 and 2 is 5
```

这个程序包含main方法和max方法。Main方法是被JVM调用的，除此之外，main方法和其它方法没什么区别。

main方法的头部是不变的，如例子所示，带修饰符public和static,返回void类型值，方法名字是main,此外带个一个String[]类型参数。String[]表明参数是字符串数组。

void 关键字

本节说明如何声明和调用一个void方法。

下面的例子声明了一个名为printGrade的方法，并且调用它来打印给定的分数。

示例

```
public class TestVoidMethod {

   public static void main(String[] args) {
      printGrade(78.5);
   }

   public static void printGrade(double score) {
      if (score >= 90.0) {
         System.out.println('A');
      }
      else if (score >= 80.0) {
         System.out.println('B');
      }
      else if (score >= 70.0) {
         System.out.println('C');
      }
      else if (score >= 60.0) {
         System.out.println('D');
      }
      else {
         System.out.println('F');
      }
   }
}
```

以上实例编译运行结果如下：

```
C
```

这里printGrade方法是一个void类型方法，它不返回值。

一个void方法的调用一定是一个语句。 所以，它被在main方法第三行以语句形式调用。就像任何以分号结束的语句一样。

通过值传递参数

调用一个方法时候需要提供参数，你必须按照参数列表指定的顺序提供。

例如，下面的方法连续n次打印一个消息：

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

示例

下面的例子演示按值传递的效果。

该程序创建一个方法，该方法用于交换两个变量。

```
public class TestPassByValue {

   public static void main(String[] args) {
      int num1 = 1;
      int num2 = 2;

      System.out.println("Before swap method, num1 is " +
                          num1 + " and num2 is " + num2);

      // 调用swap方法
      swap(num1, num2);
      System.out.println("After swap method, num1 is " +
                         num1 + " and num2 is " + num2);
   }
   /** 交换两个变量的方法 */
   public static void swap(int n1, int n2) {
      System.out.println("\tInside the swap method");
      System.out.println("\t\tBefore swapping n1 is " + n1
                           + " n2 is " + n2);
      // 交换 n1 与 n2的值
      int temp = n1;
      n1 = n2;
      n2 = temp;

      System.out.println("\t\tAfter swapping n1 is " + n1
                           + " n2 is " + n2);
   }
}
```

以上实例编译运行结果如下：

```
Before swap method, num1 is 1 and num2 is 2
        Inside the swap method
                Before swapping n1 is 1 n2 is 2
                After swapping n1 is 2 n2 is 1
After swap method, num1 is 1 and num2 is 2
```

传递两个参数调用swap方法。有趣的是，方法被调用后，实参的值并没有改变。

方法的重载

上面使用的max方法仅仅适用于int型数据。但如果你想得到两个浮点类型数据的最大值呢？

解决方法是创建另一个有相同名字但参数不同的方法，如下面代码所示：

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

如果你调用max方法时传递的是int型参数，则 int型参数的max方法就会被调用；

如果传递的事double型参数，则double类型的max方法体会被调用，这叫做方法重载；

就是说一个类的两个方法拥有相同的名字，但是有不同的参数列表。

Java编译器根据方法签名判断哪个方法应该被调用。

方法重载可以让程序更清晰易读。执行密切相关任务的方法应该使用相同的名字。

重载的方法必须拥有不同的参数列表。你不能仅仅依据修饰符或者返回类型的不同来重载方法。

变量作用域

变量的范围是程序中该变量可以被引用的部分。

方法内定义的变量被称为局部变量。

局部变量的作用范围从声明开始，直到包含它的块结束。

局部变量必须声明才可以使用。

方法的参数范围涵盖整个方法。参数实际上是一个局部变量。

for循环的初始化部分声明的变量，其作用范围在整个循环。

但循环体内声明的变量其适用范围是从它声明到循环体结束。它包含如下所示的变量声明：

```
public static void method1() {
  .
  .
  for (int i = 1; i < 10; i++) {
    .
    .
    int j;
    .
    .
    .
  }
}
```

The scope of i

The scope of j

你可以在一个方法里，不同的非嵌套块中多次声明一个具有相同的名称局部变量，但你不能在嵌套块内两次声明局部变量。

命令行参数的使用

有时候你希望运行一个程序时候再传递给它消息。这要靠传递命令行参数给main()函数实现。

命令行参数是在执行程序时候紧跟在程序名字后面的信息。

实例

下面的程序打印所有的命令行参数：

```
public class CommandLine {

   public static void main(String args[]){
      for(int i=0; i<args.length; i++){
         System.out.println("args[" + i + "]: " +
                                          args[i]);
      }
   }
}
```

如下所示，运行这个程序：

```
java CommandLine this is a command line 200 -100
```

运行结果如下：

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

构造方法

当一个对象被创建时候，构造方法用来初始化该对象。构造方法和它所在类的名字相同，但构造方法没有返回值。

通常会使用构造方法给一个类的实例变量赋初值，或者执行其它必要的步骤来创建一个完整的对象。

不管你与否自定义构造方法，所有的类都有构造方法，因为Java自动提供了一个默认构造方法，它把所有成员初始化为0。

一旦你定义了自己的构造方法，默认构造方法就会失效。

实例

下面是一个使用构造方法的例子：

```
// 一个简单的构造函数
class MyClass {
   int x;

   // 以下是构造函数
   MyClass() {
      x = 10;
   }
}
```

你可以像下面这样调用构造方法来初始化一个对象：

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

大多时候需要一个有参数的构造方法。

实例

下面是一个使用构造方法的例子：

```
// 一个简单的构造函数
class MyClass {
   int x;

   // 以下是构造函数
   MyClass(int i ) {
      x = i;
   }
}
```

你可以像下面这样调用构造方法来初始化一个对象：

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

运行结果如下：

```
10 20
```

可变参数

JDK 1.5 开始，Java支持传递同类型的可变参数给一个方法。

方法的可变参数的声明如下所示：

```
typeName... parameterName
```

在方法声明中，在指定参数类型后加一个省略号(...) 。

一个方法中只能指定一个可变参数，它必须是方法的最后一个参数。任何普通的参数必须在它之前声明。

实例

```
public class VarargsDemo {

   public static void main(String args[]) {
      // 调用可变参数的方法
      printMax(34, 3, 3, 2, 56.5);
      printMax(new double[]{1, 2, 3});
   }

   public static void printMax( double... numbers) {
   if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
   }

   double result = numbers[0];

   for (int i = 1; i <  numbers.length; i++)
      if (numbers[i] >  result)
      result = numbers[i];
      System.out.println("The max value is " + result);
   }
}
```

以上实例编译运行结果如下：

```
The max value is 56.5
The max value is 3.0
```

finalize() 方法

Java允许定义这样的方法，它在对象被垃圾收集器析构(回收)之前调用，这个方法叫做finalize( )，它用来清除回收对象。

例如，你可以使用finalize()来确保一个对象打开的文件被关闭了。

在finalize()方法里，你必须指定在对象销毁时候要执行的操作。

finalize()一般格式是：

```
protected void finalize()
{
   // 在这里终结代码
}
```

关键字protected是一个限定符，它确保finalize() 方法不会被该类以外的代码调用。

当然，Java的内存回收可以由JVM来自动完成。如果你手动使用，则可以使用上面的方法。

实例

```
public class FinalizationDemo {
    public static void main(String[] args) {
        Cake c1 = new Cake(1);
        Cake c2 = new Cake(2);
        Cake c3 = new Cake(3);

        c2 = c3 = null;
        System.gc(); //调用Java垃圾收集器
    }
}

class Cake extends Object {
    private int id;
    public Cake(int id) {
        this.id = id;
        System.out.println("Cake Object " + id + "is created");
    }

    protected void finalize() throws java.lang.Throwable {
        super.finalize();
        System.out.println("Cake Object " + id + "is disposed");
    }
}
```

运行以上代码，输出结果如下：

```
C:\1>java FinalizationDemo
Cake Object 1is created
Cake Object 2is created
Cake Object 3is created
Cake Object 3is disposed
Cake Object 2is disposed
```

Java 流(Stream)、文件(File)和IO

Java.io包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

Java.io包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

Java为I/O提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

但本节讲述最基本的和流与I/O相关的功能。我们将通过一个个例子来学习这些功能。

读取控制台输入

Java的控制台输入由Sysem.in完成。

为了获得一个绑定到控制台的字符流，你可以把System.in包装在一个BufferedReader 对象中来创建一个字符流。

下面是创建BufferedReader的基本语法：

```
BufferedReader br = new BufferedReader(new
                      InputStreamReader(System.in));
```

BufferedReader对象创建后，我们便可以使用read()方法从控制台读取一个字符，或者用readLine()方法读取一个字符串。

从控制台读取多字符输入

从BufferedReader对象读取一个字符要使用read()方法，它的语法如下：

```
int read( ) throws IOException
```

每次调用read()方法，它从输入流读取一个字符并把该字符作为整数值返回。 当流结束的时候返回-1。该方法抛出IOException。

下面的程序示范了用read()方法从控制台不断读取字符直到用户输入"q"。

```
// 使用 BufferedReader 在控制台读取字符

import java.io.*;

public class BRRead {
   public static void main(String args[]) throws IOException
   {
      char c;
      // 使用 System.in 创建 BufferedReader
      BufferedReader br = new BufferedReader(new
                         InputStreamReader(System.in));
      System.out.println("Enter characters, 'q' to quit.");
      // 读取字符
      do {
         c = (char) br.read();
         System.out.println(c);
      } while(c != 'q');
   }
}
```

以上实例编译运行结果如下:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

从控制台读取字符串

从标准输入读取一个字符串需要使用BufferedReader的readLine()方法。

它的一般格式是：

```
String readLine( ) throws IOException
```

下面的程序读取和显示字符行直到你输入了单词"end"。

```
// 使用 BufferedReader 在控制台读取字符
import java.io.*;
public class BRReadLines {
   public static void main(String args[]) throws IOException
   {
      // 使用 System.in 创建 BufferedReader
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
      String str;
      System.out.println("Enter lines of text.");
      System.out.println("Enter 'end' to quit.");
      do {
         str = br.readLine();
         System.out.println(str);
      } while(!str.equals("end"));
   }
}
```

以上实例编译运行结果如下:

```
Enter lines of text.
Enter 'end' to quit.
This is line one
This is line one
This is line two
This is line two
end
end
```

控制台输出

在此前已经介绍过，控制台的输出由 print( ) 和println( )完成。这些方法都由类PrintStream 定义，System.out是该类对象的一个引用。

PrintStream 继承了OutputStream类，并且实现了方法write()。这样，write()也可以用来往控制台写操作。

PrintStream 定义write()的最简单格式如下所示：

```
void write(int byteval)
```

该方法将byteval的低八位字节写到流中。

实例

下面的例子用write()把字符"A"和紧跟着的换行符输出到屏幕：

```
import java.io.*;

// 演示 System.out.write().
public class WriteDemo {
   public static void main(String args[]) {
      int b;
      b = 'A';
      System.out.write(b);
      System.out.write('\n');
   }
}
```

运行以上实例在输出窗口输出"A"字符

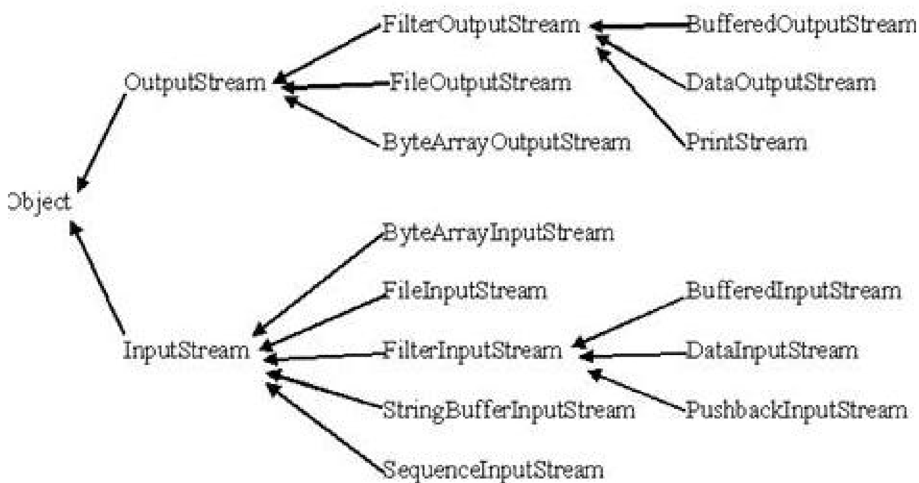
```
A
```

注意：write()方法不经常使用，因为print()和println()方法用起来更为方便。

读写文件

如前所述，一个流被定义为一个数据序列。输入流用于从源读取数据，输出流用于向目标写数据。

下图是一个描述输入流和输出流的类层次图。



下面将要讨论的两个重要的流是FileInputStream 和FileOutputStream：

FileInputStream

该流用于从文件读取数据，它的对象可以用关键字new来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

```
InputStream f = new FileInputStream("C:/java/hello");
```

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

创建了InputStream对象，就可以使用下面的方法来读取流或者进行其他的流操作。

方法	描述
public void close() throws IOException{}	关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。
protected void finalize()throws IOException {}	这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。
public int read(int r)throws IOException{}	这个方法从InputStream对象读取指定字节的数据。返回为整数值。返回下一字节数据，如果已经到结尾则返回-1。
public int read(byte[] r) throws IOException{}	这个方法从输入流读取r.length长度的字节。返回读取的字节数。如果是文件结尾则返回-1。
public int available() throws IOException{}	返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。

除了InputStream外，还有一些其他的输入流，更多的细节参考下面链接：

- ByteArrayInputStream
- DataInputStream

FileOutputStream

该类用来创建一个文件并向文件中写数据。

如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。

有两个构造方法可以用来创建FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

创建OutputStream 对象完成后，就可以使用下面的方法来写入流或者进行其他的流操作。

序号	方法及描述
public void close() throws IOException{}	关闭此文件输入流并释放与此流有关的所有系统资源。抛出IOException异常。
protected void finalize()throws IOException {}	这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出IOException异常。
public void write(int w)throws IOException{}	这个方法把指定的字节写到输出流中。
public void write(byte[] w)	把指定数组中w.length长度的字节写到OutputStream中。

除了OutputStream外，还有一些其他的输出流，更多的细节参考下面链接：

- ByteArrayOutputStream
- DataOutputStream

实例

下面是一个演示InputStream和OutputStream用法的例子：

```
import java.io.*;

public class fileStreamTest{

   public static void main(String args[]){

   try{
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x=0; x < bWrite.length ; x++){
         os.write( bWrite[x] ); // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("test.txt");
      int size = is.available();

      for(int i=0; i< size; i++){
         System.out.print((char)is.read() + "  ");
      }
      is.close();
   }catch(IOException e){
      System.out.print("Exception");
   }
   }
}
```

上面的程序首先创建文件test.txt，并把给定的数字以二进制形式写进该文件，同时输出到控制台上。

以上代码由于是二进制写入，可能存在乱码，你可以使用以下代码实例来解决乱码问题：

```
//文件名 :fileStreamTest2.java
import java.io.*;

public class fileStreamTest2{
    public static void main(String[] args) throws IOException {

        File f = new File("a.txt");
        FileOutputStream fop = new FileOutputStream(f);
        // 构建FileOutputStream对象,文件不存在会自动新建

        OutputStreamWriter writer = new OutputStreamWriter(fop, "UTF-8");
        // 构建OutputStreamWriter对象,参数可以指定编码,默认为操作系统默认编码,windows上是gbk

        writer.append("中文输入");
        // 写入到缓冲区

        writer.append("\r\n");
        //换行

        writer.append("English");
        // 刷新缓存冲,写入到文件,如果下面已经没有写入的内容了,直接close也会写入

        writer.close();
        //关闭写入流,同时会把缓冲区内容写入文件,所以上面的注释掉

        fop.close();
        // 关闭输出流,释放系统资源

        FileInputStream fip = new FileInputStream(f);
        // 构建FileInputStream对象

        InputStreamReader reader = new InputStreamReader(fip, "UTF-8");
        // 构建InputStreamReader对象,编码与写入相同

        StringBuffer sb = new StringBuffer();
        while (reader.ready()) {
            sb.append((char) reader.read());
            // 转成char加到StringBuffer对象中
        }
        System.out.println(sb.toString());
        reader.close();
        // 关闭读取流

        fip.close();
        // 关闭输入流,释放系统资源

    }
}
```

文件和I/O

还有一些关于文件和I/O的类，我们也需要知道：

- File Class(类)
- FileReader Class(类)
- FileWriter Class(类)

Java中的目录

创建目录：

File类中有两个方法可以用来创建文件夹：

- **mkdir( )**方法创建一个文件夹，成功则返回true，失败则返回false。失败表明File对象指定的路径已经存在，或者由于整个路径还不存在，该文件夹不能被创建。
- **mkdirs()**方法创建一个文件夹和它的所有父文件夹。

下面的例子创建 "/tmp/user/java/bin"文件夹：

```
import java.io.File;

public class CreateDir {
   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);
      // 现在创建目录
      d.mkdirs();
   }
}
```

编译并执行上面代码来创建目录"/tmp/user/java/bin"。

注意：Java在UNIX和Windows自动按约定分辨文件路径分隔符。如果你在Windows版本的Java中使用分隔符(/)，路径依然能够被正确解析。

读取目录

一个目录其实就是一个File对象，它包含其他文件和文件夹。

如果创建一个File对象并且它是一个目录，那么调用isDirectory( )方法会返回true。

可以通过调用该对象上的list()方法，来提取它包含的文件和文件夹的列表。

下面展示的例子说明如何使用list()方法来检查一个文件夹中包含的内容：

```
import java.io.File;

public class DirList {
   public static void main(String args[]) {
      String dirname = "/tmp";
      File f1 = new File(dirname);
      if (f1.isDirectory()) {
         System.out.println( "Directory of " + dirname);
         String s[] = f1.list();
         for (int i=0; i < s.length; i++) {
            File f = new File(dirname + "/" + s[i]);
            if (f.isDirectory()) {
               System.out.println(s[i] + " is a directory");
            } else {
               System.out.println(s[i] + " is a file");
            }
         }
      } else {
         System.out.println(dirname + " is not a directory");
    }
  }
}
```

以上实例编译运行结果如下：

```
Directory of /tmp
bin is a directory
lib is a directory
demo is a directory
test.txt is a file
README is a file
index.html is a file
include is a directory
```

Java 异常处理

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误java.lang.Error；如果你用System.out.println(11/0)，那么你是因为你用0做了除数，会抛出java.lang.ArithmeticException的异常。

异常发生的原因有很多，通常包含以下几大类：

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者JVM内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。-

要理解Java异常处理是如何工作的，你需要掌握以下三种类型的异常：

- 检查性异常：最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
- 运行时异常： 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
- 错误： 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

Exception类的层次

所有的异常类是从java.lang.Exception类继承的子类。

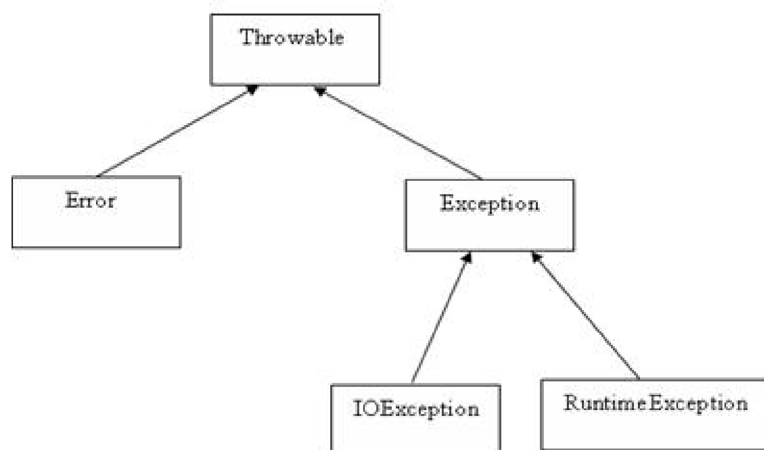
Exception类是Throwable类的子类。除了Exception类外，Throwable还有一个子类Error 。

Java程序通常不捕获错误。错误一般发生在严重故障时，它们在Java程序处理的范畴之外。

Error用来指示运行时环境发生的错误。

例如，JVM内存溢出。一般地，程序不会从错误中恢复。

异常类有两个主要的子类：IOException类和RuntimeException类。



在Java 内置类中(接下来会说明)，有大部分常用检查性和非检查性异常。

Java 内置异常类

Java 语言定义了一些异常类在java.lang标准包中。

标准运行时异常类的子类是最常见的异常类。由于java.lang包是默认加载到所有的Java程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java根据各个类库也定义了一些其他的异常，下面的表中列出了Java的非检查性异常。

异常	描述
ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了Java定义在java.lang包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 `Object` 类中的 `clone` 方法克隆对象，但该对象的类无法实现 `Cloneable` 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 `Class` 类中的 `newInstance` 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

异常方法

下面的列表是Throwable 类的主要方法:

方法	说明
public String getMessage()	返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。
public Throwable getCause()	返回一个Throwable 对象代表异常原因。
public String toString()	使用getMessage()的结果返回类的串级名字。
public void printStackTrace()	打印toString()结果和栈层次到System.err，即错误输出流。
public StackTraceElement [] getStackTrace()	返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
public Throwable fillInStackTrace()	用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。

捕获异常

使用try和catch关键字可以捕获异常。try/catch代码块放在异常可能发生的地方。

try/catch代码块中的代码称为保护代码，使用 try/catch的语法如下：

```
try
{
   // 程序代码
}catch(ExceptionName e1)
{
   //Catch 块
}
```

Catch语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try后面的catch块就会被检查。

如果发生的异常包含在catch块中，异常会被传递到该catch块，这和传递一个参数到方法是一样。

实例

下面的例子中声明有两个元素的一个数组，当代码试图访问数组的第三个元素的时候就会抛出一个异常。

```
// 文件名 : ExcepTest.java
import java.io.*;
public class ExcepTest{

   public static void main(String args[]){
      try{
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

以上代码编译运行输出结果如下：

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

多重捕获块

一个try代码块后面跟随多个catch代码块的情况就叫多重捕获。

多重捕获块的语法如下所示：

```
try{
   // 程序代码
}catch(异常类型1 异常的变量名1){
   // 程序代码
}catch(异常类型2 异常的变量名2){
   // 程序代码
}catch(异常类型2 异常的变量名2){
   // 程序代码
}
```

上面的代码段包含了3个catch块。

可以在ry语句后面添加任意数量的catch块。

如果保护代码中发生异常，异常被抛给第一个catch块。

如果抛出异常的数据类型与ExceptionType1匹配，它在这里就会被捕获。

如果不匹配，它会被传递给第二个catch块。

如此，直到异常被捕获或者通过所有的catch块。

实例

该实例展示了怎么使用多重try/catch。

```
try
{
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
   return -1;
}
```

throws/throw关键字：

如果一个方法没有捕获一个检查性异常，那么该方法必须使用throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用throw关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个RemoteException异常：

```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //Remainder of class definition
}
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出RemoteException和InsufficientFundsException：

```
import java.io.*;
public class className
{
   public void withdraw(double amount) throws RemoteException,
                              InsufficientFundsException
   {
       // Method implementation
   }
   //Remainder of class definition
}
```

finally关键字

finally关键字用来创建在try代码块后面执行的代码块。

无论是否发生异常，finally代码块中的代码总会被执行。

在finally代码块中，可以运行清理类型等收尾善后性质的语句。

finally代码块出现在catch代码块最后，语法如下：

```
 try{
    // 程序代码
 }catch(异常类型1 异常的变量名1){
    // 程序代码
 }catch(异常类型2 异常的变量名2){
    // 程序代码
 }finally{
    // 程序代码
 }
```

实例

```
public class ExcepTest{

   public static void main(String args[]){
      int a[] = new int[2];
      try{
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      finally{
         a[0] = 6;
         System.out.println("First element value: " +a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

以上实例编译运行结果如下：

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

注意下面事项：

- catch不能独立于try存在。
- 在try/catch后面添加finally块并非强制性要求的。
- try代码后不能既没catch块也没finally块。
- try, catch, finally块之间不能添加任何代码。

声明自定义异常

在Java中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

- 所有异常都必须是Throwable的子类。
- 如果希望写一个检查性异常类，则需要继承Exception类。
- 如果你想写一个运行时异常类，那么需要继承RuntimeException 类。

可以像下面这样定义自己的异常类：

```
class MyException extends Exception{
}
```

只继承Exception 类来创建的异常类是检查性异常类。

下面的InsufficientFundsException类是用户定义的异常类，它继承自Exception。

一个异常类和其它任何类一样，包含有变量和方法。

实例

```
// 文件名InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

为了展示如何使用我们自定义的异常类，

在下面的CheckingAccount 类中包含一个withdraw()方法抛出一个InsufficientFundsException异常。

```
// 文件名称 CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
   private double balance;
   private int number;
   public CheckingAccount(int number)
   {
      this.number = number;
   }
   public void deposit(double amount)
   {
      balance += amount;
   }
   public void withdraw(double amount) throws
                              InsufficientFundsException
   {
      if(amount <= balance)
      {
         balance -= amount;
      }
      else
      {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }
   public double getBalance()
   {
      return balance;
   }
   public int getNumber()
   {
      return number;
   }
}
```

下面的BankDemo程序示范了如何调用CheckingAccount类的deposit() 和withdraw()方法。

```
//文件名称 BankDemo.java
public class BankDemo
{
   public static void main(String [] args)
   {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);
      try
      {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      }catch(InsufficientFundsException e)
      {
         System.out.println("Sorry, but you are short $"
                                  + e.getAmount());
         e.printStackTrace();
      }
    }
}
```

编译上面三个文件，并运行程序BankDemo，得到结果如下所示：

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

通用异常

在Java中定义了两种类型的异常和错误。

- **JVM(Java**虚拟机)异常：**由JVM抛出的异常或错误。例如：NullPointerException类，ArrayIndexOutOfBoundsException类，ClassCastException类。
- 程序级异常：由程序或者API程序抛出的异常。例如IllegalArgumentException类，IllegalStateException类。

Java 面向对象

Java 继承

继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。继承可以理解为一个对象从另一个对象获取属性的过程。

如果类A是类B的父类，而类B是类C的父类，我们也称C是A的子类，类C是从类A继承而来的。在Java中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类

继承中最常使用的两个关键字是extends和implements。

这两个关键字的使用决定了一个对象和另一个对象是否是IS-A(是一个)关系。

通过使用这两个关键字，我们能实现一个对象获取另一个对象的属性。

所有Java的类均是由java.lang.Object类继承而来的，所以Object是所有类的祖先类，而除了Object外，所有类必须有一个父类。

通过过extends关键字可以申明一个类是继承另外一个类而来的，一般形式如下：

```
// A.java
public class A {
    private int i;
    protected int j;

    public void func() {

    }
}

// B.java
public class B extends A {
}
```

以上的代码片段说明，B由A继承而来的，B是A的子类。而A是Object的子类，这里可以不显示地声明。

作为子类，B的实例拥有A所有的成员变量，但对于private的成员变量B却没有访问权限，这保障了A的封装性。

IS-A关系

IS-A就是说:一个对象是另一个对象的一个分类。

下面是使用关键字extends实现继承。

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

基于上面的例子，以下说法是正确的：

- Animal类是Mammal类的父类。
- Animal类是Reptile类的父类。
- Mammal类和Reptile类是Animal类的子类。
- Dog类既是Mammal类的子类又是Animal类的子类。

分析以上示例中的IS-A关系，如下：

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal

因此 : Dog IS-A Animal

通过使用关键字**extends**，子类可以继承父类的除private属性外所有的属性。

我们通过使用instanceof 操作符，能够确定Mammal IS-A Animal

实例

```
public class Dog extends Mammal{

   public static void main(String args[]){

      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
true
true
true
```

介绍完**extends**关键字之后，我们再来看下**implements**关键字是怎样使用来表示IS-A关系。

Implements关键字使用在类继承接口的情况下， 这种情况不能使用关键字**extends**。

实例

```
public interface Animal {}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

instanceof 关键字

可以使用 **instanceof** 运算符来检验Mammal和dog对象是否是Animal类的一个实例。

```
interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
   public static void main(String args[]){

      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
true
true
true
```

HAS-A 关系

HAS-A代表类和它的成员之间的从属关系。这有助于代码的重用和减少代码的错误。

例子

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
    private Speed sp;
}
```

Van类和Speed类是HAS-A关系(Van有一个Speed)，这样就不用将Speed类的全部代码粘贴到Van类中了，并且Speed类也可以重复利用于多个应用程序。

在面向对象特性中，用户不必担心类的内部怎样实现。

Van类将实现的细节对用户隐藏起来，因此，用户只需要知道怎样调用Van类来完成某一功能，而不必知道Van类是自己来做还是调用其他类来做这些工作。

Java只支持单继承，也就是说，一个类不能继承多个类。

下面的做法是不合法的：

```
public class extends Animal, Mammal{}
```

Java只支持单继承（继承基本类和抽象类），但是我们可以用接口来实现（多继承接口来实现）,脚本结构如：

```
public class Apple extends Fruit implements Fruit1, Fruit2{}
```

一般我们继承基本类和抽象类用extends关键字，实现接口类的继承用implements关键字。

Java 重写(Override)与重载(Overload)

重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写！返回值和形参都不能改变。即外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。

也就是说子类能够根据需要实现父类的方法。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

```
class Animal{

   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{

   public void move(){
      System.out.println("狗可以跑和走");
   }
}

public class TestDog{

   public static void main(String args[]){
      Animal a = new Animal(); // Animal 对象
      Animal b = new Dog(); // Dog 对象

      a.move();// 执行 Animal 类的方法

      b.move();//执行 Dog 类的方法
   }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

在上面的例子中可以看到，尽管b属于Animal类型，但是它运行的是Dog类的move方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为Animal类中存在move方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

```
class Animal{

   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{

   public void move(){
      System.out.println("狗可以跑和走");
   }
   public void bark(){
      System.out.println("狗可以吠叫");
   }
}

public class TestDog{

   public static void main(String args[]){
      Animal a = new Animal(); // Animal 对象
      Animal b = new Dog(); // Dog 对象

      a.move();// 执行 Animal 类的方法
      b.move();//执行 Dog 类的方法
      b.bark();
   }
}
```

以上实例编译运行结果如下：

```
TestDog.java:30: cannot find symbol
symbol  : method bark()
location: class Animal
                b.bark();
                 ^
```

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

方写重写的规则

- 参数列表必须完全与被重写方法的相同；
- 返回类型必须完全与被重写方法的返回类型相同；
- 访问权限不能比父类中被重写的方法的访问权限更高。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。
- 父类的成员方法只能被它的子类重写。
- 声明为final的方法不能被重写。
- 声明为static的方法不能被重写，但是能够被再次声明。
- 如果一个方法不能被继承，那么该方法不能被重写。
- 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。

- 子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。
- 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
- 构造方法不能被重写。
- 如果不能继承一个方法，则不能重写这个方法。

Super关键字的使用

当需要在子类中调用父类的被重写方法时，要使用super关键字。

```
class Animal{

   public void move(){
      System.out.println("动物可以移动");
   }
}

class Dog extends Animal{

   public void move(){
      super.move(); // 应用super类的方法
      System.out.println("狗可以跑和走");
   }
}

public class TestDog{

   public static void main(String args[]){

      Animal b = new Dog(); /
      b.move(); //执行 Dog类的方法

   }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型呢？可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

只能重载构造函数

重载规则

- 被重载的方法必须改变参数列表；
- 被重载的方法可以改变返回类型；
- 被重载的方法可以改变访问修饰符；
- 被重载的方法可以声明新的或更广的检查异常；
- 方法能够在同一个类中或者在一个子类中被重载。

实例

```
public class Overloading {

    public int test(){
        System.out.println("test1");
        return 1;
    }

    public void test(int a){
        System.out.println("test2");
    }

    //以下两个参数类型顺序不同
    public String test(int a,String s){
        System.out.println("test3");
        return "returntest3";
    }

    public String test(String s,int a){
        System.out.println("test4");
        return "returntest4";
    }

    public static void main(String[] args){
        Overloading o = new Overloading();
        System.out.println(o.test());
        o.test(1);
        System.out.println(o.test(1,"test3"));
        System.out.println(o.test("test4",1));
    }
```

重写与重载之间的区别

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制（可以降低限制）

Java 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

多态性是对象多种表现形式的体现。

比如我们说"宠物"这个对象，它就有很多不同的表达或实现，比如有小猫、小狗、蜥蜴等等。那么我到宠物店说"请给我一只宠物"，服务员给我小猫、小狗或者蜥蜴都可以，我们就说"宠物"这个对象就具备多态性。

接下来让我们通过实例来了解Java的多态。

例子

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

因为Deer类具有多重继承，所以它具有多态性。以上实例解析如下：

- 一个 Deer IS-A（是一个） Animal
- 一个 Deer IS-A（是一个） Vegetarian
- 一个 Deer IS-A（是一个） Deer
- 一个 Deer IS-A（是一个）Object

在Java中，所有的对象都具有多态性，因为任何对象都能通过IS-A测试的类型和Object类。

访问一个对象的唯一方法就是通过引用型变量。

引用型变量只能有一种类型，一旦被声明，引用型变量的类型就不能被改变了。

引用型变量不仅能够被重置为其他对象，前提是这些对象没有被声明为final。还可以引用和它类型相同的或者相兼容的对象。它可以声明为类类型或者接口类型。

当我们将引用型变量应用于Deer对象的引用时，下面的声明是合法的：

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

所有的引用型变量d,a,v,o都指向堆中相同的Deer对象。

虚方法

我们将介绍在Java中，当设计类时，被重载的方法的行为怎样影响多态性。

我们已经讨论了方法的重载，也就是子类能够重载父类的方法。

当子类对象调用重载的方法时，调用的是子类的方法，而不是父类中被重载的方法。

要想调用父类中被重载的方法，则必须使用关键字super。

```
/* 文件名 : Employee.java */
public class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
      return name;
   }
   public String getAddress()
   {
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

假设下面的类继承Employee类：

```
/* 文件名 : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
   public double getSalary()
   {
       return salary;
   }
   public void setSalary(double newSalary)
   {
       if(newSalary >= 0.0)
       {
          salary = newSalary;
       }
   }
   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

现在我们仔细阅读下面的代码，尝试给出它的输出结果：

```
/* 文件名 : VirtualDemo.java */
public class VirtualDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

以上实例编译运行结果如下：

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

例子中，我们实例化了两个Salary对象。一个使用Salary引用s，另一个使用Employee引用。

编译时，编译器检查到mailCheck()方法在Salary类中的声明。

在调用s.mailCheck()时，Java虚拟机(JVM)调用Salary类的mailCheck()方法。

因为e是Employee的引用，所以调用e的mailCheck()方法则有完全不同的结果。

当编译器检查e.mailCheck()方法时，编译器检查到Employee类中的mailCheck()方法。

在编译的时候，编译器使用Employee类中的mailCheck()方法验证该语句， 但是在运行的时候，Java虚拟机(JVM)调用的是Salary类中的mailCheck()方法。

该行为被称为虚拟方法调用，该方法被称为虚拟方法。

Java中所有的方法都能以这种方式表现，借此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

Java 抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

抽象类

在Java语言中使用abstract class来定义抽象类。如下实例：

```
/* 文件名 : Employee.java */
public abstract class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public double computePay()
   {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
      return name;
   }
   public String getAddress()
   {
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

注意到该Employee类没有什么不同，尽管该类是抽象类，但是它仍然有3个成员变量，7个成员方法和1个构造方法。 现在如果你尝试如下的例子：

```
/* 文件名 : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
      /* 以下是不允许的，会引发错误 */
      Employee e = new Employee("George W.", "Houston, TX", 43);

      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

当你尝试编译AbstractDemo类时，会产生如下错误：

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

继承抽象类

我们能通过一般的方法继承Employee类：

```
/* 文件名 : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
   public double getSalary()
   {
       return salary;
   }
   public void setSalary(double newSalary)
   {
       if(newSalary >= 0.0)
       {
          salary = newSalary;
       }
   }
   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

尽管我们不能实例化一个Employee类的对象，但是如果我们实例化一个Salary类对象，该对象将从Employee类继承3个成员变量和7个成员方法。

```
/* 文件名 : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();

      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
    }
}
```

以上程序编译运行结果如下：

```
Constructing an Employee
Constructing an Employee
Call mailCheck using  Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.
```

抽象方法

如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

Abstract关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。

```
public abstract class Employee
{
   private String name;
   private String address;
   private int number;

   public abstract double computePay();

   //其余代码
}
```

声明抽象方法会造成以下两个结果：

- 如果一个类包含抽象方法，那么该类必须是抽象类。
- 任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

继承抽象方法的子类必须重载该方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终的子类都不能用来实例化对象。

如果Salary类继承了Employee类，那么它必须实现computePay()方法：

```
/* 文件名 : Salary.java */
public class Salary extends Employee
{
   private double salary; // Annual salary

   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }

   //其余代码
}
```

java 封装

在面向对象程式设计方法中，封装（英语：Encapsulation）是指，一种将抽象性函式接口的实作细节部份包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

实例

让我们来看一个java封装类的例子：

```
/* 文件名: EncapTest.java */
public class EncapTest{

   private String name;
   private String idNum;
   private int age;

   public int getAge(){
      return age;
   }

   public String getName(){
      return name;
   }

   public String getIdNum(){
      return idNum;
   }

   public void setAge( int newAge){
      age = newAge;
   }

   public void setName(String newName){
      name = newName;
   }

   public void setIdNum( String newId){
      idNum = newId;
   }
}
```

以上实例中public方法是外部类访问该类成员变量的入口。

通常情况下，这些方法被称为getter和setter方法。

因此，任何要访问类中私有成员变量的类都要通过这些getter和setter方法。

通过如下的例子说明EncapTest类的变量怎样被访问：

```
/* F文件名 : RunEncap.java */
public class RunEncap{

   public static void main(String args[]){
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName()+
                             " Age : "+ encap.getAge());
    }
}
```

以上代码编译运行结果如下:

```
Name : James Age : 20
```

Java 接口

接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在Java中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在.java结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在.class结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了static和final变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多重继承。

接口的声明

接口的声明语法格式如下：

```
[可见度] interface 接口名称 [extends 其他的类名] {
        // 声明变量
        // 抽象方法
}
```

Interface关键字用来声明一个接口。下面是接口声明的一个简单例子。

```
/* 文件名 : NameOfInterface.java */
import java.lang.*;
//引入包

public interface NameOfInterface
{
   //任何类型 final, static 字段
   //抽象方法
}
```

接口有以下特性：

- 接口是隐式抽象的，当声明一个接口的时候，不必使用**abstract**关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要**abstract**关键子。
- 接口中的方法都是公有的。

实例

```
/* 文件名 : Animal.java */
interface Animal {

   public void eat();
   public void travel();
}
```

接口的实现

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用implements关键字实现接口。在类声明中，Implements关键字放在class声明后面。

实现一个接口的语法，可以使用这个公式：

```
... implements 接口名称[, 其他接口, 其他接口..., ...] ...
```

实例

```
/* 文件名 : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

以上实例编译运行结果如下:

```
Mammal eats
Mammal travels
```

重写接口中声明的方法时，需要注意以下规则：

- 类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。
- 类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。
- 如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

- 一个类可以同时实现多个接口。
- 一个类只能继承一个类，但是能实现多个接口。
- 一个接口能继承另一个接口，这和类之间的继承比较相似。

接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用extends关键字，子接口继承父接口的方法。

下面的Sports接口被Hockey和Football接口继承：

```
// 文件名: Sports.java
public interface Sports
{
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// 文件名: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// 文件名: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

Hockey接口自己声明了四个方法，从Sports接口继承了两个方法，这样，实现Hockey接口的类需要实现六个方法。

相似的，实现Football接口的类需要实现五个方法，其中两个来自于Sports接口。

接口的多重继承

在Java中，类的多重继承是不合法，但接口允许多重继承，。

在接口的多重继承中extends关键字只需要使用一次，在其后跟着继承接口。 如下所示：

```
public interface Hockey extends Sports, Event
```

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多重继承，而 Sports及 Event 可能定义或是继承相同的方法

标记接口

最常用的继承接口是没有包含任何方法的接口。

标识接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标识接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event包中的MouseListener接口继承的java.util.EventListener接口定义如下：

```
package java.util;
public interface EventListener
{}
```

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

- 建立一个公共的父接口：

  正如EventListener接口，这是由几十个其他接口扩展的Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了EventListener接口，Java虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。

- 向一个类添加数据类型：

  这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

Java 包(package)

为了更好地组织类，Java提供了包机制，用于区别类名的命名空间。

包的作用

- 1 把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。
- 2 如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。
- 3 包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

Java使用包（package）这种机制是为了防止命名冲突，访问控制，提供搜索和定位类（class）、接口、枚举（enumerations）和注释（annotation）等。

包语句的语法格式为：

```
package pkg1[. pkg2[. pkg3…]];
```

例如,一个Something.java 文件它的内容

```
package net.java.util
public class Something{
   ...
}
```

那么它的路径应该是 net/java/Something.java 这样保存的。 package(包)的作用是把不同的java程序分类保存，更方便的被其他java程序调用。

一个包（package）可以定义为一组相互联系的类型（类、接口、枚举和注释），为这些类型提供访问保护和命名空间管理的功能。

以下是一些Java中的包：

- java.lang-打包基础的类
- java.io-包含输入输出功能的函数

开发者可以自己把一组类和接口等打包，并定义自己的package。而且在实际开发中这样做是值得提倡的，当你自己完成类的实现之后，将相关的类分组，可以让其他的编程者更容易地确定哪些类、接口、枚举和注释等是相关的。

由于package创建了新的命名空间（namespace），所以不会跟其他package中的任何名字产生命名冲突。使用包这种机制，更容易实现访问控制，并且让定位相关类更加简单。

创建包

创建package的时候，你需要为这个package取一个合适的名字。之后，如果其他的一个源文件包含了这个包提供的类、接口、枚举或者注释类型的时候，都必须将这个package的声明放在这个源文件的开头。

包声明应该在源文件的第一行，每个源文件只能有一个包声明，这个文件中的每个类型都应用于它。

如果一个源文件中没有使用包声明，那么其中的类，函数，枚举，注释等将被放在一个无名的包（unnamed package）中。

例子

让我们来看一个例子，这个例子创建了一个叫做animals的包。通常使用小写的字母来命名避免与类、接口名字的冲突。

在animals包中加入一个接口（interface）：

```
/* 文件名: Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

接下来，在同一个包中加入该接口的实现：

```
package animals;

/* 文件名 : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

然后，编译这两个文件，并把他们放在一个叫做animals的子目录中。 用下面的命令来运行：

```
$ mkdir animals
$ cp Animal.class  MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travel
```

import关键字

为了能够使用某一个包的成员，我们需要在 Java 程序中明确导入该包。使用"import"语句可完成此功能。

在 java 源文件中 import 语句应位于 package 语句之后，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

```
import package1[.package2…].(classname|*);
```

如果在一个包中，一个类想要使用本包中的另一个类，那么该包名可以省略。

例子

下面的payroll包已经包含了Employee类，接下来向payroll包中添加一个Boss类。Boss类引用Employee类的时候可以不用使用payroll前缀，Boss类的实例如下。

```
package payroll;

public class Boss
{
   public void payEmployee(Employee e)
   {
      e.mailCheck();
   }
}
```

如果Boss类不在payroll包中又会怎样？Boss类必须使用下面几种方法之一来引用其他包中的类

使用类全名描述，例如：

```
payroll.Employee
```

用import关键字引入，使用通配符"*"

```
import payroll.*;
```

使用import关键字引入Employee类

```
import payroll.Employee;
```

注意：

类文件中可以包含任意数量的import声明。import声明必须在包声明之后，类声明之前。

package的目录结构

类放在包中会有两种主要的结果：

- 包名成为类名的一部分，正如我们前面讨论的一样。
- 包名必须与相应的字节码所在的目录结构相吻合。

下面是管理你自己java中文件的一种简单方式：

将类、接口等类型的源码放在一个文本中，这个文件的名字就是这个类型的名字，并以.java作为扩展名。例如：

```
// 文件名 :  Car.java

package vehicle;

public class Car {
   // 类实现
}
```

接下来，把源文件放在一个目录中，这个目录要对应类所在包的名字。

```
....\vehicle\Car.java
```

现在，正确的类名和路径将会是如下样子：

- 类名 -> vehicle.Car
- 路径名 -> vehicle\Car.java (in windows)

通常，一个公司使用它互联网域名的颠倒形式来作为它的包名.例如：互联网域名是apple.com，所有的包名都以com.apple开头。包名中的每一个部分对应一个子目录。

例如：这个公司有一个com.apple.computers的包，这个包包含一个叫做Dell.java的源文件，那么相应的，应该有如下面的一连串子目录：

```
....\com\apple\computers\Dell.java
```

编译的时候，编译器为包中定义的每个类、接口等类型各创建一个不同的输出文件，输出文件的名字就是这个类型的名字，并加上.class作为扩展后缀。 例如：

```
// 文件名: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

现在，我们用-d选项来编译这个文件，如下：

```
$javac -d . Dell.java
```

这样会像下面这样放置编译了的文件：

```
.\com\apple\computers\Dell.class.\com\apple\computers\Ups.class
```

你可以像下面这样来导入所有 \com\apple\computers\中定义的类、接口等：

```
import com.apple.computers.*;
```

编译之后的.class文件应该和.java源文件一样，它们放置的目录应该跟包的名字对应起来。但是，并不要求.class文件的路径跟相应的.java的路径一样。你可以分开来安排源码和类的目录。

```
<path-one>\sources\com\apple\computers\Dell.java
<path-two>\classes\com\apple\computers\Dell.class
```

这样，你可以将你的类目录分享给其他的编程人员，而不用透露自己的源码。用这种方法管理源码和类文件可以让编译器和java虚拟机（JVM）可以找到你程序中使用的所有类型。

类目录的绝对路径叫做class path。设置在系统变量CLASSPATH中。编译器和java虚拟机通过将package名字加到class path后来构造.class文件的路径。

<path- two>\classes是class path，package名字是com.apple.computers,而编译器和JVM会在 <path-two>\classes\com\apple\compters中找.class文件。

一个class path可能会包含好几个路径。多路径应该用分隔符分开。默认情况下，编译器和JVM查找当前目录。JAR文件按包含Java平台相关的类，所以他们的目录默认放在了class path中。

设置CLASSPATH系统变量

用下面的命令显示当前的CLASSPATH变量：

- Windows平台（DOS 命令行下）-> C:\> set CLASSPATH
- UNIX平台（Bourne shell下）-> % echo $CLASSPATH

删除当前CLASSPATH变量内容：

- Windows平台（DOS 命令行下）-> C:\> set CLASSPATH=
- UNIX平台（Bourne shell下）-> % unset CLASSPATH; export CLASSPATH

设置CLASSPATH变量:

- Windows平台（DOS 命令行下）-> set CLASSPATH=C:\users\jack\java\classes
- UNIX平台（Bourne shell下）-> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

Java 高级教程

Java 数据结构

Java工具包提供了强大的数据结构。在Java中的数据结构主要包括以下几种接口和类：

- 枚举（Enumeration）
- 位集合（BitSet）
- 向量（Vector）
- 栈（Stack）
- 字典（Dictionary）
- 哈希表（Hashtable）
- 属性（Properties）

以上这些类是传统遗留的，在Java2中引入了一种新的框架-集合框架(Collection)，我们后面再讨论。

枚举（Enumeration）

枚举（Enumeration）接口虽然它本身不属于数据结构,但它在其他数据结构的范畴里应用很广。 枚举（The Enumeration）接口定义了一种从数据结构中取回连续元素的方式。

例如，枚举定义了一个叫nextElement 的方法，该方法用来得到一个包含多元素的数据结构的下一个元素。

关于枚举接口的更多信息，请参见枚举（Enumeration）。

位集合（BitSet）

位集合类实现了一组可以单独设置和清除的位或标志。

该类在处理一组布尔值的时候非常有用，你只需要给每个值赋值一"位"，然后对位进行适当的设置或清除，就可以对布尔值进行操作了。

关于该类的更多信息，请参见位集合（BitSet）。

向量（Vector）

向量（Vector）类和传统数组非常相似，但是Vector的大小能根据需要动态的变化。

和数组一样，Vector对象的元素也能通过索引访问。

使用Vector类最主要的好处就是在创建对象的时候不必给对象指定大小，它的大小会根据需要动态的变化。

关于该类的更多信息，请参见向量(Vector)

栈（Stack）

栈（Stack）实现了一个后进先出（LIFO）的数据结构。

你可以把栈理解为对象的垂直分布的栈，当你添加一个新元素时，就将新元素放在其他元素的顶部。

当你从栈中取元素的时候，就从栈顶取一个元素。换句话说，最后进栈的元素最先被取出。

关于该类的更多信息，请参见栈（Stack）。

字典（Dictionary）

字典（Dictionary） 类是一个抽象类，它定义了键映射到值的数据结构。

当你想要通过特定的键而不是整数索引来访问数据的时候，这时候应该使用Dictionary。

由于Dictionary类是抽象类，所以它只提供了键映射到值的数据结构，而没有提供特定的实现。

关于该类的更多信息，请参见字典（ Dictionary）。

哈希表（Hashtable）

Hashtable类提供了一种在用户定义键结构的基础上来组织数据的手段。

例如，在地址列表的哈希表中，你可以根据邮政编码作为键来存储和排序数据，而是通过人的名字。

哈希表键的具体含义完全取决于哈希表的使用情景和它包含的数据。

关于该类的更多信息，请参见哈希表（HashTable）。

属性（Properties）

Properties 继承于 Hashtable.Properties 类表示了一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

Properties 类被许多Java类使用。例如，在获取环境变量时它就作为System.getProperties()方法的返回值。

关于该类的更多信息，请参见属性（Properties）。

Java Enumeration接口

Enumeration接口中定义了一些方法，通过这些方法可以枚举（一次获得一个）对象集合中的元素。

这种传统接口已被迭代器取代，虽然Enumeration 还未被遗弃，但在现代代码中已经被很少使用了。尽管如此，它还是使用在诸如Vector和Properties这些传统类所定义的方法中，除此之外，还用在一些API类，并且在应用程序中也广泛被使用。 下表总结了一些Enumeration声明的方法：

方法	描述
boolean hasMoreElements( )	测试此枚举是否包含更多的元素。
Object nextElement( )	如果此枚举对象至少还有一个可提供的元素，则返回此枚举的下一个元素。

实例

以下实例演示了Enumeration的使用：

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester {

   public static void main(String args[]) {
      Enumeration days;
      Vector dayNames = new Vector();
      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();
      while (days.hasMoreElements()){
         System.out.println(days.nextElement());
      }
   }
}
```

以上实例编译运行结果如下：

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Java Bitset类

一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。这和位向量（vector of bits）比较类似。

这是一个传统的类，但它在Java 2中被完全重新设计。

BitSet定义了两个构造方法。

第一个构造方法创建一个默认的对象：

```
BitSet()
```

第二个方法允许用户指定初始大小。所有位初始化为0。

```
BitSet(int size)
```

BitSet中实现了Cloneable接口中定义的方法如下表所列：

方法	描述
void and(BitSet bitSet)	对此目标位 set 和参数位 set 执行逻辑与操作。
void andNot(BitSet bitSet)	清除此 BitSet 中所有的位，其相应的位在指定的 BitSet 中已设置。
int cardinality( )	返回此 BitSet 中设置为 true 的位数。
void clear( )	将此 BitSet 中的所有位设置为 false。
void clear(int index)	将索引指定处的位设置为 false。
void clear(int startIndex, int endIndex)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为 false。
Object clone( )	复制此 BitSet，生成一个与之相等的新 BitSet。
boolean equals(Object bitSet)	将此对象与指定的对象进行比较。
void flip(int index)	将指定索引处的位设置为其当前值的补码。
void flip(int startIndex, int endIndex)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的每个位设置为其当前值的补码。
boolean get(int index)	返回指定索引处的位值。
BitSet get(int startIndex, int endIndex)	返回一个新的 BitSet，它由此 BitSet 中从 fromIndex（包括）到 toIndex（不包括）范围内的位组成。

boolean intersects(BitSet bitSet)	如果指定的 BitSet 中有设置为 true 的位，并且在此 BitSet 中也将其设置为 true，则返回 ture。
boolean isEmpty( )	如果此 BitSet 中没有包含任何设置为 true 的位，则返回 ture。
int length( )	返回此 BitSet 的"逻辑大小"：BitSet 中最高设置位的索引加 1。
int nextClearBit(int startIndex)	返回第一个设置为 false 的位的索引，这发生在指定的起始索引或之后的索引上。
int nextSetBit(int startIndex)	返回第一个设置为 true 的位的索引，这发生在指定的起始索引或之后的索引上。
void or(BitSet bitSet)	对此位 set 和位 set 参数执行逻辑或操作。
void set(int index)	将指定索引处的位设置为 true。
void set(int index, boolean v)	将指定索引处的位设置为指定的值。
void set(int startIndex, int endIndex)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为 true。
void set(int startIndex, int endIndex, boolean v)	将指定的 fromIndex（包括）到指定的 toIndex（不包括）范围内的位设置为指定的值。
int size( )	返回此 BitSet 表示位值时实际使用空间的位数。
String toString( )	返回此位 set 的字符串表示形式。
void xor(BitSet bitSet)	对此位 set 和位 set 参数执行逻辑异或操作。

实例

下面的程序说明这个数据结构支持的几个方法：

```
import java.util.BitSet;

public class BitSetDemo {

  public static void main(String args[]) {
     BitSet bits1 = new BitSet(16);
     BitSet bits2 = new BitSet(16);

     // set some bits
     for(int i=0; i<16; i++) {
        if((i%2) == 0) bits1.set(i);
        if((i%5) != 0) bits2.set(i);
     }
     System.out.println("Initial pattern in bits1: ");
     System.out.println(bits1);
     System.out.println("\nInitial pattern in bits2: ");
     System.out.println(bits2);

     // AND bits
     bits2.and(bits1);
     System.out.println("\nbits2 AND bits1: ");
     System.out.println(bits2);

     // OR bits
     bits2.or(bits1);
     System.out.println("\nbits2 OR bits1: ");
     System.out.println(bits2);

     // XOR bits
     bits2.xor(bits1);
     System.out.println("\nbits2 XOR bits1: ");
     System.out.println(bits2);
  }
}
```

以上实例编译运行结果如下：

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Java Vector 类

Vector类实现了一个动态数组。和ArrayList和相似，但是两者是不同的：

- Vector是同步访问的。
- Vector包含了许多传统的方法，这些方法不属于集合框架。

Vector主要用在事先不知道数组的大小，或者只是需要一个可以改变大小的数组的情况。

Vector类支持4种构造方法。

第一种构造方法创建一个默认的向量，默认大小为10：

```
Vector()
```

第二种构造方法创建指定大小的向量。

```
Vector(int size)
```

第三种构造方法创建指定大小的向量，并且增量用incr指定. 增量表示向量每次增加的元素数目。

```
Vector(int size,int incr)
```

第四中构造方法创建一个包含集合c元素的向量：

```
Vector(Collection c)
```

除了从父类继承的方法外Vector还定义了以下方法：

方法	描述
void add(int index, Object element)	在此向量的指定位置插入指定的元素。
boolean add(Object o)	将指定元素添加到此向量的末尾。
boolean addAll(Collection c)	将指定 Collection 中的所有元素添加到此向量的末尾，按照指定 collection 的迭代器所返回的顺序添加这些元素。
boolean addAll(int index, Collection c)	在指定位置将指定 Collection 中的所有元素插入到此向量中。
void addElement(Object obj)	将指定的组件添加到此向量的末尾，将其大小增加 1。

int capacity()	返回此向量的当前容量。	
void clear()	从此向量中移除所有元素。	
Object clone()	返回向量的一个副本。	
boolean contains(Object elem)	如果此向量包含指定的元素，则返回 true。	
boolean containsAll(Collection c)	如果此向量包含指定 Collection 中的所有元素，则返回 true。	
void copyInto(Object[] anArray)	将此向量的组件复制到指定的数组中。	
Object elementAt(int index)	返回指定索引处的组件。	
Enumeration elements()	返回此向量的组件的枚举。	
void ensureCapacity(int minCapacity)	增加此向量的容量（如有必要），以确保其至少能够保存最小容量参数指定的组件数。	
boolean equals(Object o)	比较指定对象与此向量的相等性。	
Object firstElement()	返回此向量的第一个组件（位于索引 0)	处的项）。
Object get(int index)	返回向量中指定位置的元素。	
int hashCode()	返回此向量的哈希码值。	
int indexOf(Object elem)	返回此向量中第一次出现的指定元素的索引，如果此向量不包含该元素，则返回 -1。	
int indexOf(Object elem, int index)	返回此向量中第一次出现的指定元素的索引，从 index 处正向搜索，如果未找到该元素，则返回 -1。	
void insertElementAt(Object obj, int index)	将指定对象作为此向量中的组件插入到指定的 index 处。	
boolean isEmpty()	测试此向量是否不包含组件。	
Object lastElement()	返回此向量的最后一个组件。	
int lastIndexOf(Object elem)	返回此向量中最后一次出现的指定元素的索引；如果此向量不包含该元素，则返回 -1。	
int lastIndexOf(Object elem, int index)	返回此向量中最后一次出现的指定元素的索引，从 index 处逆向搜索，如果未找到该元素，则返回 -1。	
Object remove(int index)	移除此向量中指定位置的元素。	
boolean remove(Object o)	移除此向量中指定元素的第一个匹配项，如果向量不包含该元素，则元素保持不变。	

boolean removeAll(Collection c)	从此向量中移除包含在指定 Collection 中的所有元素。
void removeAllElements()	从此向量中移除全部组件，并将其大小设置为零。
boolean removeElement(Object obj)	从此向量中移除变量的第一个（索引最小的）匹配项。
void removeElementAt(int index)	删除指定索引处的组件。
protected void removeRange(int fromIndex, int toIndex)	从此 List 中移除其索引位于 fromIndex（包括）与 toIndex（不包括）之间的所有元素。
boolean retainAll(Collection c)	在此向量中仅保留包含在指定 Collection 中的元素。
Object set(int index, Object element)	用指定的元素替换此向量中指定位置处的元素。
void setElementAt(Object obj, int index)	将此向量指定 index 处的组件设置为指定的对象。
void setSize(int newSize)	设置此向量的大小。
int size()	返回此向量中的组件数。
List subList(int fromIndex, int toIndex)	返回此 List 的部分视图，元素范围为从 fromIndex（包括）到 toIndex（不包括）。
Object[] toArray()	返回一个数组，包含此向量中以恰当顺序存放的所有元素。
Object[] toArray(Object[] a)	返回一个数组，包含此向量中以恰当顺序存放的所有元素；返回数组的运行时类型为指定数组的类型。
String toString()	返回此向量的字符串表示形式，其中包含每个元素的 String 表示形式。
void trimToSize()	对此向量的容量进行微调，使其等于向量的当前大小。

实例

下面的程序说明这个集合所支持的几种方法：

```
import java.util.*;

public class VectorDemo {

   public static void main(String args[]) {
      // initial size is 3, increment is 2
      Vector v = new Vector(3, 2);
      System.out.println("Initial size: " + v.size());
      System.out.println("Initial capacity: " +
      v.capacity());
      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));
      v.addElement(new Integer(4));
      System.out.println("Capacity after four additions: " +
          v.capacity());

      v.addElement(new Double(5.45));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Double(6.08));
      v.addElement(new Integer(7));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Float(9.4));
      v.addElement(new Integer(10));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Integer(11));
      v.addElement(new Integer(12));
      System.out.println("First element: " +
         (Integer)v.firstElement());
      System.out.println("Last element: " +
         (Integer)v.lastElement());
      if(v.contains(new Integer(3)))
         System.out.println("Vector contains 3.");
      // enumerate the elements in the vector.
      Enumeration vEnum = v.elements();
      System.out.println("\nElements in vector:");
      while(vEnum.hasMoreElements())
         System.out.print(vEnum.nextElement() + " ");
      System.out.println();
   }
}
```

以上实例编译运行结果如下：

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

Java Stack 类

栈是Vector的一个子类，它实现了一个标准的后进先出的栈。

堆栈只定义了默认构造函数，用来创建一个空栈。 堆栈除了包括由Vector定义的所有方法，也定义了自己的一些方法。

```
Stack()
```

除了由Vector定义的所有方法，自己也定义了一些方法：

方法	描述
boolean empty()	测试堆栈是否为空。
Object peek( )	查看堆栈顶部的对象，但不从堆栈中移除它。
Object pop( )	移除堆栈顶部的对象，并作为此函数的值返回该对象。
Object push(Object element)	把项压入堆栈顶部。
int search(Object element)	返回对象在堆栈中的位置，以 1 为基数。

实例

下面的程序说明这个集合所支持的几种方法

```
import java.util.*;

public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);
   }

   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);
      showpush(st, 42);
      showpush(st, 66);
      showpush(st, 99);
      showpop(st);
      showpop(st);
      showpop(st);
      try {
         showpop(st);
      } catch (EmptyStackException e) {
         System.out.println("empty stack");
      }
   }
}
```

以上实例编译运行结果如下：

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

Java Dictionary 类

Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。

给出键和值，你就可以将值存储在Dictionary对象中。一旦该值被存储，就可以通过它的键来获取它。所以和Map一样， Dictionary 也可以作为一个键/值对列表。

Dictionary定义的抽象方法如下表所示：

方法	描述
Enumeration elements( )	返回此 dictionary 中值的枚举。
Object get(Object key)	返回此 dictionary 中该键所映射到的值。
boolean isEmpty( )	测试此 dictionary 是否不存在从键到值的映射。
Enumeration keys( )	返回此 dictionary 中的键的枚举。
Object put(Object key, Object value)	将指定 key 映射到此 dictionary 中指定 value。
Object remove(Object key)	从此 dictionary 中移除 key （及其相应的 value）。
int size( )	返回此 dictionary 中条目（不同键）的数量。

Dictionary类已经过时了。在实际开发中，你可以实现Map接口来获取键/值的存储功能。

Java Hashtable 接口

Hashtable是原始的java.util的一部分， 是一个Dictionary具体的实现 。

然而，Java 2 重构的Hashtable实现了Map接口，因此，Hashtable现在集成到了集合框架中。它和HashMap类很相似，但是它支持同步。

像HashMap一样，Hashtable在哈希表中存储键/值对。当使用一个哈希表，要指定用作键的对象，以及要链接到该键的值。

然后，该键经过哈希处理，所得到的散列码被用作存储在该表中值的索引。

Hashtable定义了四个构造方法。第一个是默认构造方法：

```
Hashtable()
```

第二个构造函数创建指定大小的哈希表：

```
Hashtable(int size)
```

第三个构造方法创建了一个指定大小的哈希表，并且通过fillRatio指定填充比例。

填充比例必须介于0.0和1.0之间，它决定了哈希表在重新调整大小之前的充满程度：

```
Hashtable(int size,float fillRatio)
```

第四个构造方法创建了一个以M中元素为初始化元素的哈希表。

哈希表的容量被设置为M的两倍。

```
Hashtable(Map m)
```

Hashtable中除了从Map接口中定义的方法外，还定义了以下方法：

方法	描述
void clear( )	将此哈希表清空，使其不包含任何键。
Object clone( )	创建此哈希表的浅表副本。
boolean contains(Object value)	测试此映射表中是否存在与指定值关联的键。
boolean containsKey(Object key)	测试指定对象是否为此哈希表中的键。
boolean containsValue(Object value)	如果此 Hashtable 将一个或多个键映射到此值，则返回 true。
Enumeration elements( )	返回此哈希表中的值的枚举。
Object get(Object key)	返回指定键所映射到的值，如果此映射不包含此键的映射，则返回 null. 更确切地讲，如果此映射包含满足 (key.equals(k)) 的从键 k 到值 v 的映射，则此方法返回 v；否则，返回 null。
boolean isEmpty( )	测试此哈希表是否没有键映射到值。
Enumeration keys( )	返回此哈希表中的键的枚举。
Object put(Object key, Object value)	将指定 key 映射到此哈希表中的指定 value。
void rehash( )	增加此哈希表的容量并在内部对其进行重组，以便更有效地容纳和访问其元素。
Object remove(Object key)	从哈希表中移除该键及其相应的值。
int size( )	返回此哈希表中的键的数量。
String toString( )	返回此 Hashtable 对象的字符串表示形式，其形式为 ASCII 字符 ", " （逗号加空格）分隔开的、括在括号中的一组条目。

实例

下面的程序说明这个数据结构支持的几个方法：

```
import java.util.*;

public class HashTableDemo {

   public static void main(String args[]) {
      // Create a hash map
      Hashtable balance = new Hashtable();
      Enumeration names;
      String str;
      double bal;

      balance.put("Zara", new Double(3434.34));
      balance.put("Mahnaz", new Double(123.22));
      balance.put("Ayan", new Double(1378.00));
      balance.put("Daisy", new Double(99.22));
      balance.put("Qadir", new Double(-19.08));

      // Show all balances in hash table.
      names = balance.keys();
      while(names.hasMoreElements()) {
         str = (String) names.nextElement();
         System.out.println(str + ": " +
         balance.get(str));
      }
      System.out.println();
      // Deposit 1,000 into Zara's account
      bal = ((Double)balance.get("Zara")).doubleValue();
      balance.put("Zara", new Double(bal+1000));
      System.out.println("Zara's new balance: " +
      balance.get("Zara"));
   }
}
```

以上实例编译运行结果如下：

```
Qadir: -19.08
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0

Zara's new balance: 4434.34
```

Java Properties 接口

Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

Properties 类被许多Java类使用。例如，在获取环境变量时它就作为System.getProperties()方法的返回值。

Properties 定义如下实例变量.这个变量持有一个Properties对象相关的默认属性列表。

```
Properties defaults;
```

Properties类定义了两个构造方法. 第一个构造方法没有默认值。

```
Properties()
```

第二个构造方法使用propDefault 作为默认值。两种情况下，属性列表都为空：

```
Properties(Properties propDefault)
```

除了从Hashtable中所定义的方法，Properties定义了以下方法：

方法	描述
String getProperty(String key)	用指定的键在此属性列表中搜索属性。
String getProperty(String key, String defaultProperty)	用指定的键在属性列表中搜索属性。
void list(PrintStream streamOut)	将属性列表输出到指定的输出流。
void list(PrintWriter streamOut)	将属性列表输出到指定的输出流。
void load(InputStream streamIn) throws IOException	从输入流中读取属性列表（键和元素对）。
Enumeration propertyNames( )	按简单的面向行的格式从输入字符流中读取属性列表（键和元素对）。
Object setProperty(String key, String value)	调用 Hashtable 的方法 put。
void store(OutputStream streamOut, String description)	以适合使用 load(InputStream)方法加载到 Properties 表中的格式，将此 Properties 表中的属性列表（键和元素对）写入输出流。

实例

下面的程序说明这个数据结构支持的几个方法：

```
import java.util.*;

public class PropDemo {

   public static void main(String args[]) {
      Properties capitals = new Properties();
      Set states;
      String str;

      capitals.put("Illinois", "Springfield");
      capitals.put("Missouri", "Jefferson City");
      capitals.put("Washington", "Olympia");
      capitals.put("California", "Sacramento");
      capitals.put("Indiana", "Indianapolis");

      // Show all states and capitals in hashtable.
      states = capitals.keySet(); // get set-view of keys
      Iterator itr = states.iterator();
      while(itr.hasNext()) {
         str = (String) itr.next();
         System.out.println("The capital of " +
            str + " is " + capitals.getProperty(str) + ".");
      }
      System.out.println();

      // look for state not in list -- specify default
      str = capitals.getProperty("Florida", "Not Found");
      System.out.println("The capital of Florida is "
          + str + ".");
   }
}
```

以上实例编译运行结果如下：

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```

Java 集合框架

早在Java 2中之前，Java就提供了特设类。比如：Dictionary, Vector, Stack, 和Properties这些类用来存储和操作对象组。

虽然这些类都非常有用，但是它们缺少一个核心的，统一的主题。由于这个原因，使用Vector类的方式和使用Properties类的方式有着很大不同。

集合框架被设计成要满足以下几个目标。

- 该框架必须是高性能的。基本集合（动态数组，链表，树，哈希表）的实现也必须是高效的。
- 该框架允许不同类型的集合，以类似的方式工作，具有高度的互操作性。
- 对一个集合的扩展和适应必须是简单的。

为此，整个集合框架就围绕一组标准接口而设计。你可以直接使用这些接口的标准实现，诸如： LinkedList, HashSet, 和 TreeSet等,除此之外你也可以通过这些接口实现自己的集合。

集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：

- 接口：是代表集合的抽象数据类型。接口允许集合独立操纵其代表的细节。在面向对象的语言，接口通常形成一个层次。
- 实现（类）：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构。
- 算法：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个Map接口和类。Map里存储的是键/值对。尽管Map不是collections，但是它们完全整合在集合中。

集合接口

集合框架定义了一些接口。本节提供了每个接口的概述：

接口	描述
Collection	接口 允许你使用一组对象，是Collection层次结构的根接口。
List	接口 继承于**Collection**和一个 List实例存储一个有序集合的元素。
Set	继承于 **Collection**，是一个不包含重复元素的集合。
SortedSet	继承于Set保存有序的集合。
Map	将唯一的键映射到值。
Map.Entry	描述在一个Map中的一个元素（键/值对）。是一个Map的内部类。
SortedMap	继承于Map，使Key保持在升序排列。
Enumeration	这是一个传统的接口和定义的方法，通过它可以枚举（一次获得一个）对象集合中的元素。这个传统接口已被迭代器取代。

集合类

Java提供了一套实现了Collection接口的标准集合类。其中一些是具体类，这些类可以直接拿来使用，而另外一些是抽象类，提供了接口的部分实现。

标准集合类汇总于下表：

类	描述
AbstractCollection	实现了大部分的集合接口。
AbstractList	继承于AbstractCollection 并且实现了大部分List接口。
AbstractSequentialList	继承于 AbstractList ，提供了对数据元素的链式访问而不是随机访问。
LinkedList	继承于 AbstractSequentialList，实现了一个链表。
ArrayList	通过继承AbstractList，实现动态数组。
AbstractSet	继承于AbstractCollection 并且实现了大部分Set接口。
HashSet	继承了AbstractSet，并且使用一个哈希表。
LinkedHashSet	具有可预知迭代顺序的 `Set` 接口的哈希表和链接列表实现。
TreeSet	继承于AbstractSet，使用元素的自然顺序对元素进行排序.
AbstractMap	实现了大部分的Map接口。
HashMap	继承了HashMap，并且使用一个哈希表。
TreeMap	继承了AbstractMap，并且使用一颗树。
WeakHashMap	继承AbstractMap类，使用弱密钥的哈希表。
LinkedHashMap	继承于HashMap，使用元素的自然顺序对元素进行排序.
IdentityHashMap	继承AbstractMap类，比较文档时使用引用相等。

在前面的教程中已经讨论通过java.util包中定义的类，如下所示：

类	描述
Vector	Vector类实现了一个动态数组。和ArrayList和相似，但是两者是不同的。
Stack	栈是Vector的一个子类，它实现了一个标准的后进先出的栈。
Dictionary	Dictionary 类是一个抽象类，用来存储键/值对，作用和Map类相似。
Hashtable	Hashtable是原始的java.util的一部分， 是一个Dictionary具体的实现 。
Properties	Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。
BitSet	一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

一个Bitset类创建一种特殊类型的数组来保存位值。BitSet中数组大小会随需要增加。

集合算法

集合框架定义了几种算法，可用于集合和映射。这些算法被定义为集合类的静态方法。

在尝试比较不兼容的类型时，一些方法能够抛出 ClassCastException异常。当试图修改一个不可修改的集合时，抛出UnsupportedOperationException异常。

集合定义三个静态的变量：EMPTY_SET EMPTY_LIST，EMPTY_MAP的。这些变量都不可改变。

算法描述
Collection Algorithms 这里是一个列表中的所有算法实现。

如何使用迭代器

通常情况下，你会希望遍历一个集合中的元素。例如，显示集合中的每个元素。

做到这一点最简单的方法是采用一个迭代器，它是一个对象，实现了Iterator 接口或ListIterator接口。

迭代器，使你能够通过循环来得到或删除集合的元素。ListIterator继承了Iterator，以允许双向遍历列表和修改元素。

这里通过实例列出Iterator和listIterator接口提供的所有方法。

迭代器方法描述
使用 Java Iterator

如何使用比较器

TreeSet和TreeMap的按照排序顺序来存储元素. 然而，这是通过比较器来精确定义按照什么样的排序顺序。

这个接口可以让我们以不同的方式来排序一个集合。

比较器方法描述
使用 Java Comparator 这里通过实例列出Comparator接口提供的所有方法

总结

Java集合框架为程序员提供了预先包装的数据结构和算法来操纵他们。

集合是一个对象，可容纳其他对象的引用。集合接口声明对每一种类型的集合可以执行的操作。

集合框架的类和接口均在java.util包中。

Java 泛型

如果我们只写一个排序方法，就能够对整形数组、字符串数组甚至支持排序的任何类型的数组进行排序，这该多好啊。

Java泛型方法和泛型类支持程序员使用一个方法指定一组相关方法，或者使用一个类指定一组相关的类型。

Java泛型（generics）是JDK 5中引入的一个新特性,泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。

使用Java泛型的概念，我们可以写一个泛型方法来对一个对象数组排序。然后，调用该泛型方法来对整型数组、浮点数数组、字符串数组等进行排序。

泛型方法

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

- 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的<E>）。
- 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
- 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
- 泛型方法方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像int,double,char的等）。

实例

下面的例子演示了如何使用泛型方法打印不同字符串的元素：

```
public class GenericMethodTest
{
   // 泛型方法 printArray
   public static < E > void printArray( E[] inputArray )
   {
      // 输出数组元素
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // 创建不同类型数组： Integer, Double 和 Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // 传递一个整型数组

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // 传递一个双精度型数组

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // 传递一个字符型型数组
    }
}
```

编译以上代码，运行结果如下所示：

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

有界的类型参数:

可能有时候，你会想限制那些被允许传递到一个类型参数的类型种类范围。例如，一个操作数字的方法可能只希望接受Number或者Number子类的实例。这就是有界类型参数的目的。

要声明一个有界的类型参数，首先列出类型参数的名称，后跟extends关键字，最后紧跟它的上界。

实例

下面的例子演示了"extends"如何使用在一般意义上的意思"extends"（类）或者"implements"（接口）。该例子中的泛型方法返回三个可比较对象的最大值。

```
public class MaximumTest
{
   // 比较三个值并返回最大值
   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
   {
      T max = x; // 假设x是初始最大值
      if ( y.compareTo( max ) > 0 ){
         max = y; //y 更大
      }
      if ( z.compareTo( max ) > 0 ){
         max = z; // 现在 z 更大
      }
      return max; // 返回最大对象
   }
   public static void main( String args[] )
   {
      System.out.printf( "Max of %d, %d and %d is %d\n\n",
                   3, 4, 5, maximum( 3, 4, 5 ) );

      System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
                   6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

      System.out.printf( "Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum( "pear", "apple", "orange" ) );
   }
}
```

编译以上代码，运行结果如下所示：

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

实例

如下实例演示了我们如何定义一个泛型类:

```
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n", integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

编译以上代码，运行结果如下所示：

```
Integer Value :10

String Value :Hello World
```

Java序列化

Java 提供了一种对象序列化的机制，该机制中，一个对象可以被表示为一个字节序列，该字节序列包括该对象的数据、有关对象的类型的信息和存储在对象中数据的类型。

将序列化对象写入文件之后，可以从文件中读取出来，并且对它进行反序列化，也就是说，对象的类型信息、对象的数据，还有对象中的数据类型可以用来在内存中新建对象。

整个过程都是Java虚拟机（JVM）独立的，也就是说，在一个平台上序列化的对象可以在另一个完全不同的平台上反序列化该对象。

类ObjectInputStream 和ObjectOutputStream是高层次的数据流，它们包含序列化和反序列化对象的方法。

ObjectOutputStream 类包含很多写方法来写各种数据类型，但是一个特别的方法例外：

```
public final void writeObject(Object x) throws IOException
```

上面的方法序列化一个对象，并将它发送到输出流。相似的ObjectInputStream 类包含如下反序列化一个对象的方法：

```
public final Object readObject() throws IOException,
                                        ClassNotFoundException
```

该方法从流中取出下一个对象，并将对象反序列化。它的返回值为Object，因此，你需要将它转换成合适的数据类型。

为了演示序列化在Java中是怎样工作的，我将使用之前教程中提到的Employee类，假设我们定义了如下的Employee类，该类实现了Serializable 接口。

```
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + name
                           + " " + address);
   }
}
```

请注意，一个类的对象要想序列化成功，必须满足两个条件：

该类必须实现 java.io.Serializable 对象。

该类的所有属性必须是可序列化的。如果有一个属性不是可序列化的，则该属性必须注明是短暂的。

如果你想知道一个Java标准类是否是可序列化的，请查看该类的文档。检验一个类的实例是否能序列化十分简答，只需要查看该类有没有实现java.io.Serializable接口。

序列化对象

ObjectOutputStream 类用来序列化一个对象，如下的SerializeDemo例子实例化了一个Employee对象，并将该对象序列化到一个文件中。

该程序执行后，就创建了一个名为employee.ser文件。该程序没有任何输出，但是你可以通过代码研读来理解程序的作用。

注意：当序列化一个对象到文件时，按照Java的标准约定是给文件一个.ser扩展名。

```
import java.io.*;

public class SerializeDemo
{
   public static void main(String [] args)
   {
      Employee e = new Employee();
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
      e.SSN = 11122333;
      e.number = 101;
      try
      {
         FileOutputStream fileOut =
         new FileOutputStream("/tmp/employee.ser");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
         fileOut.close();
         System.out.printf("Serialized data is saved in /tmp/employee.ser");
      }catch(IOException i)
      {
          i.printStackTrace();
      }
   }
}
```

反序列化对象

下面的DeserializeDemo程序反序列化在SerializeDemo程序中创建Employee对象。

```
import java.io.*;
public class DeserializeDemo
{
   public static void main(String [] args)
   {
      Employee e = null;
      try
      {
         FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      }catch(IOException i)
      {
         i.printStackTrace();
         return;
      }catch(ClassNotFoundException c)
      {
         System.out.println("Employee class not found");
         c.printStackTrace();
         return;
      }
      System.out.println("Deserialized Employee...");
      System.out.println("Name: " + e.name);
      System.out.println("Address: " + e.address);
      System.out.println("SSN: " + e.SSN);
      System.out.println("Number: " + e.number);
    }
}
```

以上程序编译运行结果如下所示：

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

这里要注意以下要点：

readObject() 方法中的try/catch代码块尝试捕获 ClassNotFoundException异常。对于JVM可以反序列化对象，它必须是能够找到字节码的类。如果JVM在反序列化对象的过程中找不到该类，则抛出一个 ClassNotFoundException异常。

注意，readObject()方法的返回值被转化成Employee引用。

当对象被序列化时，属性SSN的值为111222333，但是因为该属性是短暂的，该值没有被发送到输出流。所以反序列化后Employee对象的SSN属性为0。

Java 网络编程

网络编程是指编写运行在多个设备（计算机）的程序，这些设备都通过网络连接起来。

java.net包中J2SE的API包含有类和接口，它们提供低层次的通信细节。你可以直接使用这些类和接口，来专注于解决问题，而不用关注通信细节。

java.net包中提供了两种常见的网络协议的支持：

- **TCP**： TCP是传输控制协议的缩写，它保障了两个应用程序之间的可靠通信。通常用于互联网协议，被称TCP / IP。
- **UDP**:UDP是用户数据报协议的缩写，一个无连接的协议。提供了应用程序之间要发送的数据的数据包。

本教程主要讲解以下两个主题。

- **Socket** 编程: 这是使用最广泛的网络概念，它已被解释地非常详细
- **URL** 处理: 这部分会在另外的篇幅里讲，点击这里更详细地了解在Java语言中的URL处理。

Socket 编程

套接字使用TCP提供了两台计算机之间的通信机制。 客户端程序创建一个套接字，并尝试连接服务器的套接字。

当连接建立时，服务器会创建一个Socket对象。客户端和服务器现在可以通过对Socket对象的写入和读取来进行进行通信。

java.net.Socket类代表一个套接字，并且java.net.ServerSocket类为服务器程序提供了一种来监听客户端，并与他们建立连接的机制。

以下步骤在两台计算机之间使用套接字建立TCP连接时会出现：

- 服务器实例化一个ServerSocket对象，表示通过服务器上的端口通信。
- 服务器调用 ServerSocket类 的accept（）方法，该方法将一直等待，直到客户端连接到服务器上给定的端口。
- 服务器正在等待时，一个客户端实例化一个Socket对象，指定服务器名称和端口号来请求连接。
- Socket类的构造函数试图将客户端连接到指定的服务器和端口号。如果通信被建立，则在客户端创建一个Socket对象能够与服务器进行通信。
- 在服务器端，accept()方法返回服务器上一个新的socket引用，该socket连接到客户端的socket。

连接建立后，通过使用I/O流在进行通信。每一个socket都有一个输出流和一个输入流。客户端的输出流连接到服务器端的输入流，而客户端的输入流连接到服务器端的输出流。

TCP是一个双向的通信协议，因此数据可以通过两个数据流在同一时间发送.以下是一些类提供的一套完整的有用的方法来实现sockets。

ServerSocket 类的方法

服务器应用程序通过使用java.net.ServerSocket类以获取一个端口,并且侦听客户端请求。

ServerSocket类有四个构造方法：

方法	描述
public ServerSocket(int port) throws IOException	创建绑定到特定端口的服务器套接字。
public ServerSocket(int port, int backlog) throws IOException	利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。
public ServerSocket(int port, int backlog, InetAddress address) throws IOException	使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器。
public ServerSocket() throws IOException	创建非绑定服务器套接字。

创建非绑定服务器套接字。 如果ServerSocket构造方法没有抛出异常，就意味着你的应用程序已经成功绑定到指定的端口，并且侦听客户端请求。

这里有一些ServerSocket类的常用方法：

方法	描述
public int getLocalPort()	返回此套接字在其上侦听的端口。
public Socket accept() throws IOException	侦听并接受到此套接字的连接。
public void setSoTimeout(int timeout)	通过指定超时值启用/禁用 SO_TIMEOUT，以毫秒为单位。
public void bind(SocketAddress host, int backlog)	将 ServerSocket 绑定到特定地址（IP 地址和端口号）。

Socket 类的方法

java.net.Socket类代表客户端和服务器都用来互相沟通的套接字。客户端要获取一个Socket对象通过实例化 ，而 服务器获得一个Socket对象则通过accept()方法的返回值。

Socket类有五个构造方法.

方法	描述
public Socket(String host, int port) throws UnknownHostException, IOException.	创建一个流套接字并将其连接到指定主机上的指定端口号。
public Socket(InetAddress host, int port) throws IOException	创建一个流套接字并将其连接到指定 IP 地址的指定端口号。
public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.	创建一个套接字并将其连接到指定远程主机上的指定远程端口。
public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.	创建一个套接字并将其连接到指定远程地址上的指定远程端口。
public Socket()	通过系统默认类型的 SocketImpl 创建未连接套接字

当Socket构造方法返回，并没有简单的实例化了一个Socket对象，它实际上会尝试连接到指定的服务器和端口。

下面列出了一些感兴趣的方法，注意客户端和服务器端都有一个Socket对象，所以无论客户端还是服务端都能够调用这些方法。

方法	描述
public void connect(SocketAddress host, int timeout) throws IOException	将此套接字连接到服务器，并指定一个超时值。
public InetAddress getInetAddress()	返回套接字连接的地址。
public int getPort()	返回此套接字连接到的远程端口。
public int getLocalPort()	返回此套接字绑定到的本地端口。
public SocketAddress getRemoteSocketAddress()	返回此套接字连接的端点的地址，如果未连接则返回 null。
public InputStream getInputStream() throws IOException	返回此套接字的输入流。
public OutputStream getOutputStream() throws IOException	返回此套接字的输出流。
public void close() throws IOException	关闭此套接字。

InetAddress 类的方法

这个类表示互联网协议(IP)地址。下面列出了Socket编程时比较有用的方法：

方法	描述
static InetAddress getByAddress(byte[] addr)	在给定原始 IP 地址的情况下，返回 InetAddress 对象。
static InetAddress getByAddress(String host, byte[] addr)	根据提供的主机名和 IP 地址创建 InetAddress。
static InetAddress getByName(String host)	在给定主机名的情况下确定主机的 IP 地址。
String getHostAddress()	返回 IP 地址字符串（以文本表现形式）。
String getHostName()	获取此 IP 地址的主机名。
static InetAddress getLocalHost()	返回本地主机。
String toString()	将此 IP 地址转换为 String。

Socket 客户端实例

如下的GreetingClient 是一个客户端程序，改程序通过socket连接到服务器并发送一个问候，然后等待一个响应。

```
// 文件名 GreetingClient.java
<pre>
import java.net.*;
import java.io.*;

public class GreetingClient
{
   public static void main(String [] args)
   {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try
      {
         System.out.println("Connecting to " + serverName
                             + " on port " + port);
         Socket client = new Socket(serverName, port);
         System.out.println("Just connected to "
                      + client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out =
                       new DataOutputStream(outToServer);

         out.writeUTF("Hello from "
                      + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in =
                        new DataInputStream(inFromServer);
         System.out.println("Server says " + in.readUTF());
         client.close();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Socket 服务器实例

如下的GreetingServer 程序是一个服务器端应用程序，改程序使用Socket来监听一个指定的端口。

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

像下面一样开启客户端：

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Java 发送邮件

使用Java应用程序发送E-mail十分简单，但是首先你应该在你的机器上安装JavaMail API 和Java Activation Framework (JAF) 。

你可以在 JavaMail (Version 1.2) 下载最新的版本。

你可以再 在JAF (Version 1.1.1)下载最新的版本。

下载并解压这些文件，最上层文件夹你会发现很多的jar文件。你需要将mail.jar和activation.jar 添加到你的CLASSPATH中。

如果你使用第三方邮件服务器如QQ的SMTP服务器，可查看文章底部用户认证完整的实例。

发送一封简单的 E-mail

下面是一个发送简单E-mail的例子。假设你的localhost已经连接到网络。

```
// 文件名 SendEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail
{
   public static void main(String [] args)
   {
      // 收件人电子邮箱
      String to = "abcd@gmail.com";

      // 发件人电子邮箱
      String from = "web@gmail.com";

      // 指定发送邮件的主机为 localhost
      String host = "localhost";

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认session对象
      Session session = Session.getDefaultInstance(properties);

      try{
         // 创建默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // Set Subject: 头部头字段
         message.setSubject("This is the Subject Line!");

         // 设置消息体
         message.setText("This is actual message");

         // 发送消息
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

编译并运行这个程序来发送一封简单的E-mail：

```
$ java SendEmail
Sent message successfully....
```

如果你想发送一封e-mail给多个收件人，那么使用下面的方法来指定多个收件人ID：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

下面是对于参数的描述：

- **type:**要被设置为TO, CC 或者BCC. 这里CC 代表抄送、BCC 代表秘密抄送y. 举例：*Message.RecipientType.TO*
- **addresses:** 这是email ID的数组。在指定电子邮件ID时，你将需要使用InternetAddress()方法。

发送一封HTML E-mail

下面是一个发送HTML E-mail的例子。假设你的localhost已经连接到网络。

和上一个例子很相似，除了我们要使用setContent()方法来通过第二个参数为"text/html"，来设置内容来指定要发送HTML内容。

```
// 文件名 SendHTMLEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendHTMLEmail
{
   public static void main(String [] args)
   {

      // 收件人电子邮箱
      String to = "abcd@gmail.com";

      // 发件人电子邮箱
      String from = "web@gmail.com";

      // 指定发送邮件的主机为 localhost
      String host = "localhost";

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象。
      Session session = Session.getDefaultInstance(properties);

      try{
         // 创建默认的 MimeMessage 对象。
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                                  new InternetAddress(to));

         // Set Subject: 头字段
         message.setSubject("This is the Subject Line!");

         // 发送 HTML 消息, 可以插入html标签
         message.setContent("<h1>This is actual message</h1>",
                            "text/html" );

         // 发送消息
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

编译并运行此程序来发送HTML e-mail：

```
$ java SendHTMLEmail
Sent message successfully....
```

发送带有附件的 E-mail

下面是一个发送带有附件的 E-mail的例子。假设你的localhost已经连接到网络。

```
// 文件名 SendFileEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendFileEmail
{
   public static void main(String [] args)
   {

      // 收件人电子邮箱
      String to = "abcd@gmail.com";

      // 发件人电子邮箱
      String from = "web@gmail.com";

      // 指定发送邮件的主机为 localhost
      String host = "localhost";

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象。
      Session session = Session.getDefaultInstance(properties);

      try{
         // 创建默认的 MimeMessage 对象。
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // Set Subject: 头字段
         message.setSubject("This is the Subject Line!");

         // 创建消息部分
         BodyPart messageBodyPart = new MimeBodyPart();

         // 消息
         messageBodyPart.setText("This is message body");

         // 创建多重消息
         Multipart multipart = new MimeMultipart();

         // 设置文本消息部分
         multipart.addBodyPart(messageBodyPart);

         // 附件部分
         messageBodyPart = new MimeBodyPart();
         String filename = "file.txt";
         DataSource source = new FileDataSource(filename);
         messageBodyPart.setDataHandler(new DataHandler(source));
         messageBodyPart.setFileName(filename);
         multipart.addBodyPart(messageBodyPart);

         // 发送完整消息
         message.setContent(multipart );

         //   发送消息
```

```
            Transport.send(message);
            System.out.println("Sent message successfully....");
         }catch (MessagingException mex) {
            mex.printStackTrace();
         }
      }
   }
```

编译并运行你的程序来发送一封带有附件的邮件。

```
$ java SendFileEmail
Sent message successfully....
```

用户认证部分

如果需要提供用户名和密码给e-mail服务器来达到用户认证的目的，你可以通过如下设置来完成：

```
 props.put("mail.smtp.auth", "true");
 props.setProperty("mail.user", "myuser");
 props.setProperty("mail.password", "mypwd");
```

e-mail其他的发送机制和上述保持一致。

需要用户名密码验证邮件发送实例:

本实例以QQ邮件服务器为例，你需要在登录QQ邮箱后台在"设置"=》账号中开启POP3/SMTP服务，如下图所示：

POP3/IMAP/SMTP/Exchange/CardDAV/CalDAV服务

开启服务：
- ☑ POP3/SMTP服务 (如何使用 Foxmail 等软件收发邮件？)
- ☑ IMAP/SMTP服务 (什么是 IMAP，它又是如何设置？)
- ☑ Exchange服务 (什么是Exchange，它又是如何设置？)
- ☑ CardDAV/CalDAV服务 (什么是CardDAV/CalDAV，它又是如何设置？)

(POP3/IMAP/SMTP/CardDAV/CalDAV服务均支持SSL连接。如何设置？)

Java 代码如下：

```
// 需要用户名密码邮件发送实例
//文件名 SendEmail2.java
//本实例以QQ邮箱为例，你需要在qq后台设置

import java.util.Properties;

import javax.mail.Authenticator;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendEmail2
{
   public static void main(String [] args)
   {
      // 收件人电子邮箱
      String to = "xxx@qq.com";

      // 发件人电子邮箱
      String from = "xxx@qq.com";

      // 指定发送邮件的主机为 localhost
      String host = "smtp.qq.com";  //QQ 邮件服务器

      // 获取系统属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      properties.put("mail.smtp.auth", "true");
      // 获取默认session对象
      Session session = Session.getDefaultInstance(properties,new Authenticator(){
        public PasswordAuthentication getPasswordAuthentication()
        {
         return new PasswordAuthentication("xxx@qq.com", "qq邮箱密码"); //发件人邮件用户名、密
        }
       });

      try{
         // 创建默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);

         // Set From: 头部头字段
         message.setFrom(new InternetAddress(from));

         // Set To: 头部头字段
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // Set Subject: 头部头字段
         message.setSubject("This is the Subject Line!");

         // 设置消息体
         message.setText("This is actual message");

         // 发送消息
         Transport.send(message);
         System.out.println("Sent message successfully....from w3cschool.cc");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

Java 多线程编程

Java给多线程编程提供了内置的支持。一个多线程程序包含两个或多个能并发运行的部分。程序的每一部分都称作一个线程，并且每个线程定义了一个独立的执行路径。

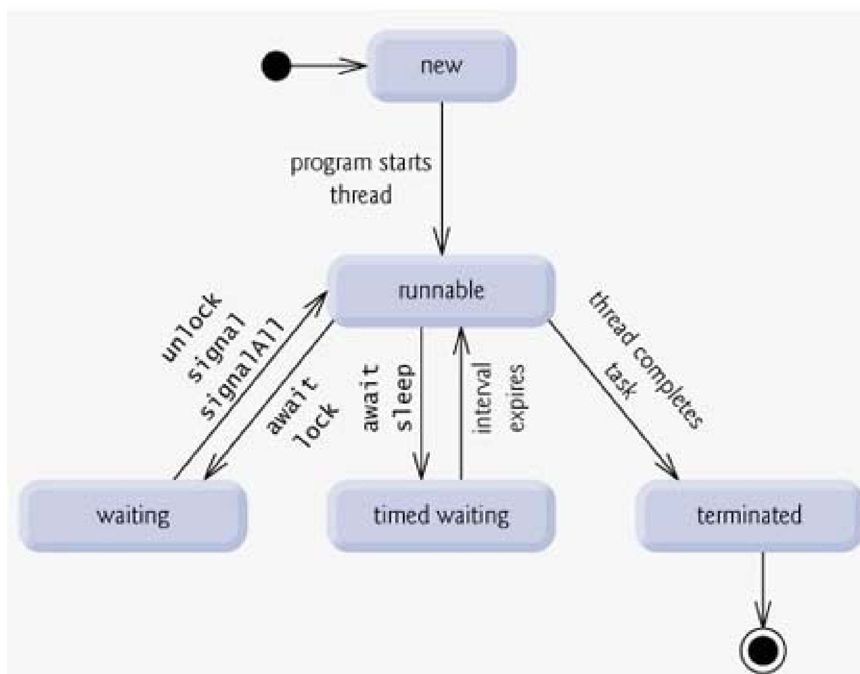
多线程是多任务的一种特别的形式。多线程比多任务需要更小的开销。

这里定义和线程相关的另一个术语：进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的非守候线程都结束运行后才能结束。

多线程能满足程序员编写非常有效率的程序来达到充分利用CPU的目的，因为CPU的空闲时间能够保持在最低限度。

一个线程的生命周

线程经过其生命周期的各个阶段。下图显示了一个线程完整的生命周期。



- 新状态: 一个新产生的线程从新状态开始了它的生命周期。它保持这个状态知道程序start这个线程。
- 运行状态:当一个新状态的线程被start以后，线程就变成可运行状态，一个线程在此状态下被认为是开始执行其任务
- 就绪状态:当一个线程等待另外一个线程执行一个任务的时候，该线程就进入就绪状态。当另一个线程给就绪状态的线程发送信号时，该线程才重新切换到运行状态。
- 休眠状态: 由于一个线程的时间片用完了，该线程从运行状态进入休眠状态。当时间间隔

到期或者等待的时间发生了，该状态的线程切换到运行状态。

- 终止状态: 一个运行状态的线程完成任务或者其他终止条件发生，该线程就切换到终止状态。

线程的优先级

每一个Java线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。Java优先级在MIN_PRIORITY（1）和MAX_PRIORITY（10）之间的范围内。默认情况下，每一个线程都会分配一个优先级NORM_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器时间。然而，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

创建一个线程

Java提供了两种创建线程方法：

- 通过实现Runable接口；
- 通过继承Thread类本身。

通过实现Runnable接口来创建线程

创建一个线程，最简单的方法是创建一个实现Runnable接口的类。

为了实现Runnable，一个类只需要执行一个方法调用run()，声明如下：

```
publicvoid run()
```

你可以重写该方法，重要的是理解的run()可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现Runnable接口的类之后，你可以在类中实例化一个线程对象。

Thread定义了几个构造方法，下面的这个是我们经常使用的：

```
Thread(Runnable threadOb,String threadName);
```

这里，threadOb 是一个实现Runnable 接口的类的实例，并且 threadName指定新线程的名字。

新线程创建之后，你调用它的start()方法它才会运行。

```
void start();
```

实例

下面是一个创建线程并开始让它执行的实例：

```
// 创建一个新的线程
class NewThread implements Runnable {
   Thread t;
   NewThread() {
      // 创建第二个新线程
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
      t.start(); // 开始线程
   }

   // 第二个线程入口
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            // 暂停线程
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
     }
     System.out.println("Exiting child thread.");
   }
}

public class ThreadDemo {
   public static void main(String args[]) {
      new NewThread(); // 创建一个新线程
      try {
         for(int i = 5; i > 0; i--) {
           System.out.println("Main Thread: " + i);
           Thread.sleep(100);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

编译以上程序运行结果如下：

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

通过继承Thread来创建线程

创建一个线程的第二种方法是创建一个新的类，该类继承Thread类，然后创建一个该类的实例。

继承类必须重写run()方法，该方法是新线程的入口点。它也必须调用start()方法才能执行。

实例

```
// 通过继承 Thread 创建线程
class NewThread extends Thread {
   NewThread() {
      // 创建第二个新线程
      super("Demo Thread");
      System.out.println("Child thread: " + this);
      start(); // 开始线程
   }

   // 第二个线程入口
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
                            // 让线程休眠一会
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
   }
}

public class ExtendThread {
   public static void main(String args[]) {
      new NewThread(); // 创建一个新线程
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);
            Thread.sleep(100);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

编译以上程序运行结果如下：

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Thread 方法

下表列出了Thread类的一些重要方法：

方法	描述	
public void start()	使该线程开始执行；**Java**	虚拟机调用该线程的 run 方法。
public void run()	如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。	
public final void setName(String name)	改变线程名称，使之与参数 name 相同。	
public final void setPriority(int priority)	更改线程的优先级。	
public final void setDaemon(boolean on)	将该线程标记为守护线程或用户线程。	
public final void join(long millisec)	等待该线程终止的时间最长为 millis 毫秒。	
public void interrupt()	中断线程。	
public final boolean isAlive()	测试线程是否处于活动状态。	

测试线程是否处于活动状态。 上述方法是被Thread对象调用的。下面的方法是Thread类的静态方法。

方法	描述
public static void yield()	暂停当前正在执行的线程对象，并执行其他线程。
public static void sleep(long millisec)	在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。
public static boolean holdsLock(Object x)	当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。
public static Thread currentThread()	返回对当前正在执行的线程对象的引用。
public static void dumpStack()	将当前线程的堆栈跟踪打印至标准错误流。

实例

如下的ThreadClassDemo 程序演示了Thread类的一些方法：

```
// 文件名 : DisplayMessage.java
// 通过实现 Runnable 接口创建线程
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

```
// 文件名 : GuessANumber.java
// 通过继承 Thread 类创建线程

public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()
                       + " guesses " + guess);
          counter++;
      }while(guess != number);
      System.out.println("** Correct! " + this.getName()
                       + " in " + counter + " guesses.**");
   }
}
```

```
// 文件名 : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(hello);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try
      {
         thread3.join();
      }catch(InterruptedException e)
      {
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);

           thread4.start();
      System.out.println("main() is ending...");
   }
}
```

运行结果如下，每一次运行的结果都不一样。

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Thread-2 guesses 27
Hello
** Correct! Thread-2 in 102 guesses.**
Hello
Starting thread4...
Hello
Hello
..........remaining result produced.
```

线程的几个主要概念:

在多线程编程时，你需要了解以下几个概念：

- 线程同步
- 线程间通信
- 线程死锁
- 线程控制：挂起、停止和恢复

多线程的使用

有效利用多线程的关键是理解程序是并发执行而不是串行执行的。例如：程序中有两个子系统需要并发执行，这时候就需要利用多线程编程。

通过对多线程的使用，可以编写出非常高效的程序。不过请注意，如果你创建太多的线程，程序执行的效率实际上是降低了，而不是提升了。

请记住，上下文的切换开销也很重要，如果你创建了太多的线程，CPU花费在上下文的切换的时间将多于执行程序的时间！

Java Applet基础

applet是一种Java程序。它一般运行在支持Java的Web浏览器内。因为它有完整的Java API支持,所以applet是一个全功能的Java应用程序。

如下所示是独立的Java应用程序和applet程序之间重要的不同：

- Java中applet类继承了 java.applet.Applet类
- Applet类没有定义main()，所以一个 Applet程序不会调用main()方法，
- Applets被设计为嵌入在一个HTML页面。
- 当用户浏览包含Applet的HTML页面，Applet的代码就被下载到用户的机器上。
- 要查看一个applet需要JVM。 JVM可以是Web浏览器的一个插件，或一个独立的运行时环境。
- 用户机器上的JVM创建一个applet类的实例，并调用Applet生命周期过程中的各种方法。
- Applets有Web浏览器强制执行的严格的安全规则，applet的安全机制被称为沙箱安全。
- applet需要的其他类可以用Java归档（JAR）文件的形式下载下来。

Applet的生命周期

Applet类中的四个方法给你提供了一个框架，你可以再该框架上开发小程序：

- **init:** 该方法的目的是为你的applet提供所需的任何初始化。在Applet标记内的param标签被处理后调用该方法。
- **start:** 浏览器调用init方法后，该方法被自动调用。每当用户从其他页面返回到包含Applet的页面时，则调用该方法。
- **stop:**当用户从包含applet的页面移除的时候，该方法自动被调用。因此，可以在相同的applet中反复调用该方法。
- **destroy:** 此方法仅当浏览器正常关闭时调用。因为applets只有在HTML网页上有效，所以你不应该在用户离开包含Applet的页面后遗漏任何资源.
- **paint:** 该方法在start()方法之后立即被调用，或者在applet需要重绘在浏览器的时候调用。paint()方法实际上继承于java.awt。

"Hello, World" Applet:

下面是一个简单的Applet程序HelloWorldApplet.java:

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet
{
   public void paint (Graphics g)
   {
      g.drawString ("Hello World", 25, 50);
   }
}
```

这些import语句将以下类导入到我们的applet类中：

```
java.applet.Applet.
java.awt.Graphics.
```

没有这些import语句，Java编译器就识别不了Applet和Graphics类。

Applet 类

每一个applet都是java.applet.Applet 类的子类，基础的Applet类提供了供衍生类调用的方法,以此来得到浏览器上下文的信息和服务。

这些方法做了如下事情：

- 得到applet的参数
- 得到包含applet的HTML文件的网络位置
- 得到applet类目录的网络位置
- 打印浏览器的状态信息
- 获取一张图片
- 获取一个音频片段
- 播放一个音频片段
- 调整此 applet 的大小

除此之外，Applet类还提供了一个接口，该接口供Viewer或浏览器来获取applet的信息，并且来控制applet的执行。

Viewer可能是：

- 请求applet作者、版本和版权的信息
- 请求applet识别的参数的描述
- 初始化applet
- 销毁applet
- 开始执行applet
- 结束执行applet

Applet类提供了对这些方法的默认实现，这些方法可以在需要的时候重写。

"Hello，World"applet都是按标准编写的。唯一被重写的方法是paint方法。

Applet的调用

applet是一种Java程序。它一般运行在支持Java的Web浏览器内。因为它有完整的Java API支持,所以applet是一个全功能的Java应用程序。

<applet>标签是在HTML文件中嵌入applet的基础。以下是一个调用"Hello World"applet的例子；

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width="320" height="120">
If your browser was Java-enabled, a "Hello, World"
message would appear here.
</applet>
<hr>
</html>
```

注意: 你可以参照HTML Applet标签来更多的了解从HTML中调用applet的方法。

<applet>标签的属性指定了要运行的Applet类。Width和height用来指定applet运行面板的初始大小。applet必须使用</applet>标签来关闭。

如果applet接受参数，那么参数的值需要在<param>标签里添加，该标签位于<applet>和</applet>之间。浏览器忽略了applet标签之间的文本和其他标签。

不支持Java的浏览器不能执行<applet>和</applet>。因此，在标签之间显示并且和applet没有关系的任何东西，在不支持的Java的浏览器里是可见的。

Viewer或者浏览器在文档的位置寻找编译过的Java代码，要指定文档的路径，得使用<applet>标签的codebase属性指定。

如下所示：

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

如果applet所在一个包中而不是默认包，那么所在的包必须在code属性里指定，例如：

```
<applet code="mypackage.subpackage.TestApplet.class"
           width="320" height="120">
```

获得applet参数

下面的例子演示了如何使用一个applet响应来设置文件中指定的参数。该Applet显示了一个黑色棋盘图案和第二种颜色。

第二种颜色和每一列的大小通过文档中的applet的参数指定。

CheckerApplet 在init()方法里得到它的参数。也可以在paint()方法里得到它的参数。然而，在applet开始得到值并保存了设置，而不是每一次刷新的时候都得到值，这样是很方便，并且高效的。

applet viewer或者浏览器在applet每次运行的时候调用init()方法。在加载applet之后，Viewer立即调用init()方法（Applet.init()什么也没做），重写该方法的默认实现，添加一些自定义的初始化代码。

Applet.getParameter()方法通过给出参数名称得到参数值。如果得到的值是数字或者其他非字符数据，那么必须解析为字符串类型。

下例是CheckerApplet.java的梗概：

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet
{
   int squareSize = 50;// 初始化默认大小
   public void init () {}
   private void parseSquareSize (String param) {}
   private Color parseColor (String param) {}
   public void paint (Graphics g) {}
}
```

下面是CheckerApplet类的init()方法和私有的parseSquareSize()方法：

```
public void init ()
{
   String squareSizeParam = getParameter ("squareSize");
   parseSquareSize (squareSizeParam);
   String colorParam = getParameter ("color");
   Color fg = parseColor (colorParam);
   setBackground (Color.black);
   setForeground (fg);
}
private void parseSquareSize (String param)
{
   if (param == null) return;
   try {
      squareSize = Integer.parseInt (param);
   }
   catch (Exception e) {
     // 保留默认值
   }
}
```

该applet调用parseSquareSize()，来解析squareSize参数。parseSquareSize()调用了库方法Integer. parseInt()，该方法将一个字符串解析为一个整数，当参数无效的时候，Integer.parseInt()抛出异常。

因此，parseSquareSize()方法也是捕获异常的，并不允许applet接受无效的输入。

Applet调用parseColor()方法将颜色参数解析为一个Color值。parseColor()方法做了一系列字符串的比较，来匹配参数的值和预定义颜色的名字。你需要实现这些方法来使applet工作。

指定applet参数

如下的例子是一个HTML文件，其中嵌入了CheckerApplet类。HTML文件通过使用<param>标签的方法给applet指定了两个参数。

```
<html>
<title>Checkerboard Applet</title>
<hr>
<applet code="CheckerApplet.class" width="480" height="320">
<param name="color" value="blue">
<param name="squaresize" value="30">
</applet>
<hr>
</html>
```

注意: 参数名字大小写不敏感。

应用程序转换成Applet

将图形化的Java应用程序（是指，使用AWT的应用程序和使用java程序启动器启动的程序）转换成嵌入在web页面里的applet是很简单的。

下面是将应用程序转换成applet的几个步骤：

- 编写一个HTML页面，该页面带有能加载applet代码的标签。
- 编写一个JApplet类的子类，将该类设置为public。否则，applet不能被加载。
- 消除应用程序的main()方法。不要为应用程序构造框架窗口，因为你的应用程序要显示在浏览器中。
- 将应用程序中框架窗口的构造方法里的初始化代码移到applet的init()方法中，你不必显示的构造applet对象，浏览器将通过调用init()方法来实例化一个对象。
- 移除对setSize()方法的调用，对于applet来讲，大小已经通过HTML文件里的width和height参数设定好了。
- 移除对 setDefaultCloseOperation()方法的调用。Applet不能被关闭，它随着浏览器的退出而终止。
- 如果应用程序调用了setTitle()方法，消除对该方法的调用。applet不能有标题栏。（当然你可以给通过html的title标签给网页自身命名）
- 不要调用setVisible(true),applet是自动显示的。

事件处理

Applet类从Container类继承了许多事件处理方法。Container类定义了几个方法，例如：processKeyEvent()和processMouseEvent()，用来处理特别类型的事件，还有一个捕获所有事件的方法叫做processEvent。

为了响应一个事件，applet必须重写合适的事件处理方法。

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet
                             implements MouseListener {

    StringBuffer strBuffer;

    public void init() {
         addMouseListener(this);
         strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
         //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
                      getWidth() - 1,
                      getHeight() - 1);

         //display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }

    public void mouseEntered(MouseEvent event) {
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }

    public void mouseClicked(MouseEvent event) {
         addItem("mouse clicked! ");
    }
}
```

如下调用该applet：

```
<html>
<title>Event Handling</title>
<hr>
<applet code="ExampleEventHandling.class"
width="300" height="300">
</applet>
<hr>
</html>
```

最开始运行，applet显示 "initializing the applet. Starting the applet."，然后你一点击矩形框，就会显示"mouse clicked" 。

显示图片

applet能显示GIF,JPEG,BMP等其他格式的图片。为了在applet中显示图片，你需要使用java.awt.Graphics类的drawImage()方法。

如下实例演示了显示图片的所有步骤：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet
{
  private Image image;
  private AppletContext context;
  public void init()
  {
      context = this.getAppletContext();
      String imageURL = this.getParameter("image");
      if(imageURL == null)
      {
         imageURL = "java.jpg";
      }
      try
      {
         URL url = new URL(this.getDocumentBase(), imageURL);
         image = context.getImage(url);
      }catch(MalformedURLException e)
      {
         e.printStackTrace();
         // Display in browser status bar
         context.showStatus("Could not load image!");
      }
   }
   public void paint(Graphics g)
   {
      context.showStatus("Displaying image");
      g.drawImage(image, 0, 0, 200, 84, null);
      g.drawString("www.javalicense.com", 35, 100);
   }
}
```

如下调用该applet：

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="300" height="200">
<param name="image" value="java.jpg">
</applet>
<hr>
</html>
```

播放音频

Applet能通过使用java.applet包中的AudioClip接口播放音频。AudioClip接口定义了三个方法：

- **public void play():** 从一开始播放音频片段一次。
- **public void loop():** 循环播放音频片段
- **public void stop():** 停止播放音频片段

为了得到AudioClip对象，你必须调用Applet类的getAudioClip()方法。无论URL指向的是否是一个真实的音频文件，该方法都会立即返回结果。

直到要播放音频文件时，该文件才会下载下来。

如下实例演示了播放音频的所有步骤：

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet
{
   private AudioClip clip;
   private AppletContext context;
   public void init()
   {
      context = this.getAppletContext();
      String audioURL = this.getParameter("audio");
      if(audioURL == null)
      {
         audioURL = "default.au";
      }
      try
      {
         URL url = new URL(this.getDocumentBase(), audioURL);
         clip = context.getAudioClip(url);
      }catch(MalformedURLException e)
      {
         e.printStackTrace();
         context.showStatus("Could not load audio file!");
      }
   }
   public void start()
   {
      if(clip != null)
      {
         clip.loop();
      }
   }
   public void stop()
   {
      if(clip != null)
      {
         clip.stop();
      }
   }
}
```

如下调用applet：

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
<hr>
```

你可以使用你电脑上的test.wav来测试上面的实例。

Java 文档注释

Java只是三种注释方式。前两种分别是// 和/* */，第三种被称作说明注释，它以/** 开始，以 */结束。

说明注释允许你在程序中嵌入关于程序的信息。你可以使用javadoc工具软件来生成信息，并输出到HTML文件中。

说明注释，是你更加方面的记录你的程序的信息。

javadoc 标签

javadoc工具软件识别以下标签：

标签	描述	示例
@author	标识一个类的作者	@author description
@deprecated	指名一个过期的类或成员	@deprecated description
{@docRoot}	指明当前文档根目录的路径	Directory Path
@exception	标志一个类抛出的异常	@exception exception-name explanation
{@inheritDoc}	从直接父类继承的注释	Inherits a comment from the immediate surperclass.
{@link}	插入一个到另一个主题的链接	{@link name text}
{@linkplain}	插入一个到另一个主题的链接，但是该链接显示纯文本字体	Inserts an in-line link to another topic.
@param	说明一个方法的参数	@param parameter-name explanation
@return	说明返回值类型	@return explanation
@see	指定一个到另一个主题的链接	@see anchor
@serial	说明一个序列化属性	@serial description
@serialData	说明通过writeObject( ) 和 writeExternal( )方法写的数据	@serialData description
@serialField	说明一个ObjectStreamField组件	@serialField name type description
@since	标记当引入一个特定的变化时	@since release
@throws	和 @exception标签一样.	The @throws tag has the same meaning as the @exception tag.
{@value}	显示常量的值，该常量必须是static属性。	Displays the value of a constant, which must be a static field.
@version	指定类的版本	@version info

文档注释

在开始的/**之后，第一行或几行是关于类、变量和方法的主要描述.

之后，你可以包含一个或多个何种各样的@标签。每一个@标签必须在一个新行的开始或者在一行的开始紧跟星号(*).

多个相同类型的标签应该放成一组。例如，如果你有三个@see标签，可以将它们一个接一个的放在一起。

下面是一个类的说明注释的示例：

```
/*** This class draws a bar chart.
* @author Zara Ali
* @version 1.2
*/
```

javadoc输出什么

javadoc工具将你Java程序的源代码作为输入，输出一些包含你程序注释的HTML文件。

每一个类的信息将在独自的HTML文件里。javadoc也可以输出继承的树形结构和索引。

由于javadoc的实现不同，工作也可能不同，你需要检查你的Java开发系统的版本等细节，选择合适的Javadoc版本。

实例

下面是一个使用说明注释的简单实例。注意每一个注释都在它描述的项目的前面。

在经过javadoc处理之后，SquareNum类的注释将在SquareNum.html中找到。

```
import java.io.*;

/**
* This class demonstrates documentation comments.
* @author Ayan Amhed
* @version 1.2
*/
public class SquareNum {
   /**
   * This method returns the square of num.
   * This is a multiline description. You can use
   * as many lines as you like.
   * @param num The value to be squared.
   * @return num squared.
   */
   public double square(double num) {
      return num * num;
   }
   /**
   * This method inputs a number from the user.
   * @return The value input as a double.
   * @exception IOException On input error.
   * @see IOException
   */
   public double getNumber() throws IOException {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader inData = new BufferedReader(isr);
      String str;
      str = inData.readLine();
      return (new Double(str)).doubleValue();
   }
   /**
   * This method demonstrates square().
   * @param args Unused.
   * @return Nothing.
   * @exception IOException On input error.
   * @see IOException
   */
   public static void main(String args[]) throws IOException
   {
      SquareNum ob = new SquareNum();
      double val;
      System.out.println("Enter value to be squared: ");
      val = ob.getNumber();
      val = ob.square(val);
      System.out.println("Squared value is " + val);
   }
}
```

如下，使用javadoc工具处理SquareNum.java文件：

```
$ javadoc SquareNum.java
Loading source file SquareNum.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0_13
Building tree for all the packages and classes...
Generating SquareNum.html...
SquareNum.java:39: warning - @return tag cannot be used\
                      in method with void return type.
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...
1 warning
$
```

Servlet 教程

Servlet 简介

Servlet 是什么？

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

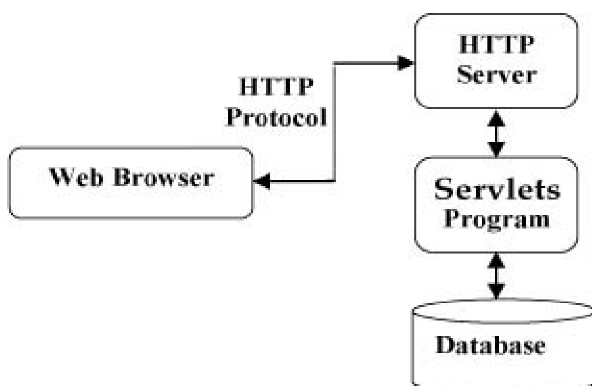
使用 Servlet，您可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页。

Java Servlet 通常情况下与使用 CGI（Common Gateway Interface，公共网关接口）实现的程序可以达到异曲同工的效果。但是相比于 CGI，Servlet 有以下几点优势：

- 性能明显更好。
- Servlet 在 Web 服务器的地址空间内执行。这样它就没有必要再创建一个单独的进程来处理每个客户端请求。
- Servlet 是独立于平台的，因为它们是用 Java 编写的。
- 服务器上的 Java 安全管理器执行了一系列限制，以保护服务器计算机上的资源。因此，Servlet 是可信的。
- Java 类库的全部功能对 Servlet 来说都是可用的。它可以通过 sockets 和 RMI 机制与 applets、数据库或其他软件进行交互。

Servlet 架构

下图显示了 Servlet 在 Web 应用程序中的位置。



Servlet 任务

Servlet 执行以下主要任务：

- 读取客户端（浏览器）发送的显式的数据。这包括网页上的 HTML 表单，或者也可以是

来自 applet 或自定义的 HTTP 客户端程序的表单。

- 读取客户端（浏览器）发送的隐式的 HTTP 请求数据。这包括 cookies、媒体类型和浏览器能理解的压缩格式等等。
- 处理数据并生成结果。这个过程可能需要访问数据库，执行 RMI 或 CORBA 调用，调用 Web 服务，或者直接计算得出对应的响应。
- 发送显式的数据（即文档）到客户端（浏览器）。该文档的格式可以是多种多样的，包括文本文件（HTML 或 XML）、二进制文件（GIF 图像）、Excel 等。
- 发送隐式的 HTTP 响应到客户端（浏览器）。这包括告诉浏览器或其他客户端被返回的文档类型（例如 HTML），设置 cookies 和缓存参数，以及其他类似的任务。

Servlet 包

Java Servlet 是运行在带有支持 Java Servlet 规范的解释器的 web 服务器上的 Java 类。

Servlet 可以使用 **javax.servlet** 和 **javax.servlet.http** 包创建，它是 Java 企业版的标准组成部分，Java 企业版是支持大型开发项目的 Java 类库的扩展版本。

这些类实现 Java Servlet 和 JSP 规范。在写本教程的时候，二者相应的版本分别是 Java Servlet 2.5 和 JSP 2.1。

Java Servlet 就像任何其他的 Java 类一样已经被创建和编译。在您安装 Servlet 包并把它们添加到您的计算机上的 Classpath 类路径中之后，您就可以通过 JDK 的 Java 编译器或任何其他编译器来编译 Servlet。

下一步呢？

接下来，本教程会带你一步一步地设置您的 Servlet 环境，以便开始后续的 Servlet 使用。因此，请系紧您的安全带，随我们一起开始 Servlet 的学习之旅吧！相信您会很喜欢这个教程的。

Servlet 环境设置

开发环境是您可以开发、测试、运行 Servlet 的地方。

就像任何其他的 Java 程序，您需要通过使用 Java 编译器 **javac** 编译 Servlet，在编译 Servlet 应用程序后，将它部署在配置的环境中以便测试和运行。

这个开发环境设置包括以下步骤：

设置 Java 开发工具包（Java Development Kit）

这一步涉及到下载 Java 软件开发工具包（SDK，即 Software Development Kit），并适当地设置 PATH 环境变量。

您可以从 Oracle 的 Java 网站下载 SDK：Java SE Downloads。

一旦您下载了 SDK，请按照给定的指令来安装和配置设置。最后，设置 PATH 和 JAVA_HOME 环境变量指向包含 java 和 javac 的目录，通常分别为 java_install_dir/bin 和 java_install_dir。

如果您运行的是 Windows，并把 SDK 安装在 C:\jdk1.5.0_20 中，则需要在您的 C:\autoexec.bat 文件中放入下列的行：

```
set PATH=C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME=C:\jdk1.5.0_20
```

或者，在 Windows NT/2000/XP 中，您也可以用鼠标右键单击"我的电脑"，选择"属性"，再选择"高级"，"环境变量"。然后，更新 PATH 的值，按下"确定"按钮。

在 Unix（Solaris、Linux 等）上，如果 SDK 安装在 /usr/local/jdk1.5.0_20 中，并且您使用的是 C shell，则需要在您的 .cshrc 文件中放入下列的行：

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

另外，如果您使用集成开发环境（IDE，即 Integrated Development Environment），比如 Borland JBuilder、Eclipse、IntelliJ IDEA 或 Sun ONE Studio，编译并运行一个简单的程序，以确认该 IDE 知道您安装的 Java 路径。

设置 Web 服务器：Tomcat

在市场上有许多 Web 服务器支持 Servlet。有些 Web 服务器是免费下载的，Tomcat 就是其中的一个。

Apache Tomcat 是一款 Java Servlet 和 JavaServer Pages 技术的开源软件实现，可以作为测试 Servlet 的独立服务器，而且可以集成到 Apache Web 服务器。下面是在电脑上安装 Tomcat 的步骤：

- 从 http://tomcat.apache.org/ 上下载最新版本的 Tomcat。
- 一旦您下载了 Tomcat，解压缩到一个方便的位置。例如，如果您使用的是 Windows，则解压缩到 C:\apache-tomcat-5.5.29 中，如果您使用的是 Linux/Unix，则解压缩到 /usr/local/apache-tomcat-5.5.29 中，并创建 CATALINA_HOME 环境变量指向这些位置。

在 Windows 上，可以通过执行下面的命令来启动 Tomcat：

```
%CATALINA_HOME%\bin\startup.bat

or

C:\apache-tomcat-5.5.29\bin\startup.bat
```

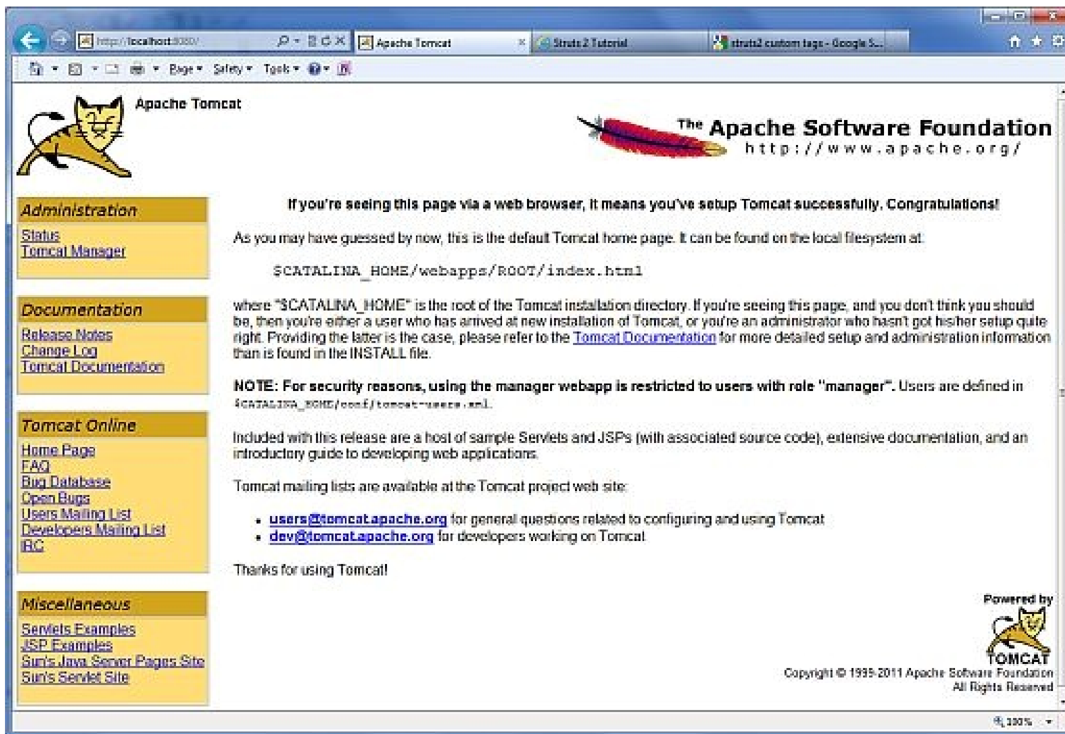
在 Unix（Solaris、Linux 等） 上，可以通过执行下面的命令来启动 Tomcat：

```
$CATALINA_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-5.5.29/bin/startup.sh
```

Tomcat 启动后，可以通过在浏览器地址栏输入 **http://localhost:8080/** 访问 Tomcat 中的默认应用程序。如果一切顺利，那么会显示以下结果：



有关配置和运行 Tomcat 的进一步信息可以查阅应用程序安装的文档，或者可以访问 Tomcat 网站：http://tomcat.apache.org。

在 Windows 上，可以通过执行下面的命令来停止 Tomcat：

```
C:\apache-tomcat-5.5.29\bin\shutdown
```

在 Unix（Solaris、Linux 等） 上，可以通过执行下面的命令来停止 Tomcat：

```
/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

设置 CLASSPATH

由于 Servlet 不是 Java 平台标准版的组成部分，所以您必须为编译器指定 Servlet 类的路径。

如果您运行的是 Windows，则需要在您的 C:\autoexec.bat 文件中放入下列的行：

```
set CATALINA=C:\apache-tomcat-5.5.29
set CLASSPATH=%CATALINA%\common\lib\servlet-api.jar;%CLASSPATH%
```

或者，在 Windows NT/2000/XP 中，您也可以用鼠标右键单击"我的电脑"，选择"属性"，再选择"高级"，"环境变量"。然后，更新 CLASSPATH 的值，按下"确定"按钮。

在 Unix（Solaris、Linux 等）上，如果您使用的是 C shell，则需要在您的 .cshrc 文件中放入下列的行：

```
setenv CATALINA=/usr/local/apache-tomcat-5.5.29
setenv CLASSPATH $CATALINA/common/lib/servlet-api.jar:$CLASSPATH
```

注意：假设您的开发目录是 C:\ServletDevel（在 Windows 上）或 /user/ServletDevel（在 UNIX 上），那么您还需要在 CLASSPATH 中添加这些目录，添加方式与上面的添加方式类似。

Servlet 生命周期

Servlet 生命周期可被定义为从创建直到毁灭的整个过程。以下是 Servlet 遵循的过程：

- Servlet 通过调用 **init ()** 方法进行初始化。
- Servlet 调用 **service()** 方法来处理客户端的请求。
- Servlet 通过调用 **destroy()** 方法终止（结束）。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

现在让我们详细讨论生命周期的方法。

init() 方法

init 方法被设计成只调用一次。它在第一次创建 Servlet 时被调用，在后续每次用户请求时不再调用。因此，它是用于一次性初始化，就像 Applet 的 init 方法一样。

Servlet 创建于用户第一次调用对应于该 Servlet 的 URL 时，但是您也可以指定 Servlet 在服务器第一次启动时被加载。

当用户调用一个 Servlet 时，就会创建一个 Servlet 实例，每一个用户请求都会产生一个新的线程，适当的时候移交给 doGet 或 doPost 方法。init() 方法简单地创建或加载一些数据，这些数据将被用于 Servlet 的整个生命周期。

init 方法的定义如下：

```
public void init() throws ServletException {
  // 初始化代码...
}
```

service() 方法

service() 方法是执行实际任务的主要方法。Servlet 容器（即 Web 服务器）调用 service() 方法来处理来自客户端（浏览器）的请求，并把格式化的响应写回给客户端。

每次服务器接收到一个 Servlet 请求时，服务器会产生一个新的线程并调用服务。service() 方法检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet、doPost、doPut，doDelete 等方法。

下面是该方法的特征：

```
public void service(ServletRequest request,
                    ServletResponse response)
      throws ServletException, IOException{
}
```

service() 方法由容器调用，service 方法在适当的时候调用 doGet、doPost、doPut、doDelete 等方法。所以，您不用对 service() 方法做任何动作，您只需要根据来自客户端的请求类型来重载 doGet() 或 doPost() 即可。

doGet() 和 doPost() 方法是每次服务请求中最常用的方法。下面是这两种方法的特征。

doGet() 方法

GET 请求来自于一个 URL 的正常请求，或者来自于一个未指定 METHOD 的 HTML 表单，它由 doGet() 方法处理。

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet 代码
}
```

doPost() 方法

POST 请求来自于一个特别指定了 METHOD 为 POST 的 HTML 表单，它由 doPost() 方法处理。

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet 代码
}
```

destroy() 方法

destroy() 方法只会被调用一次，在 Servlet 生命周期结束时被调用。destroy() 方法可以让您的 Servlet 关闭数据库连接、停止后台线程、把 Cookie 列表或点击计数器写入到磁盘，并执行其他类似的清理活动。

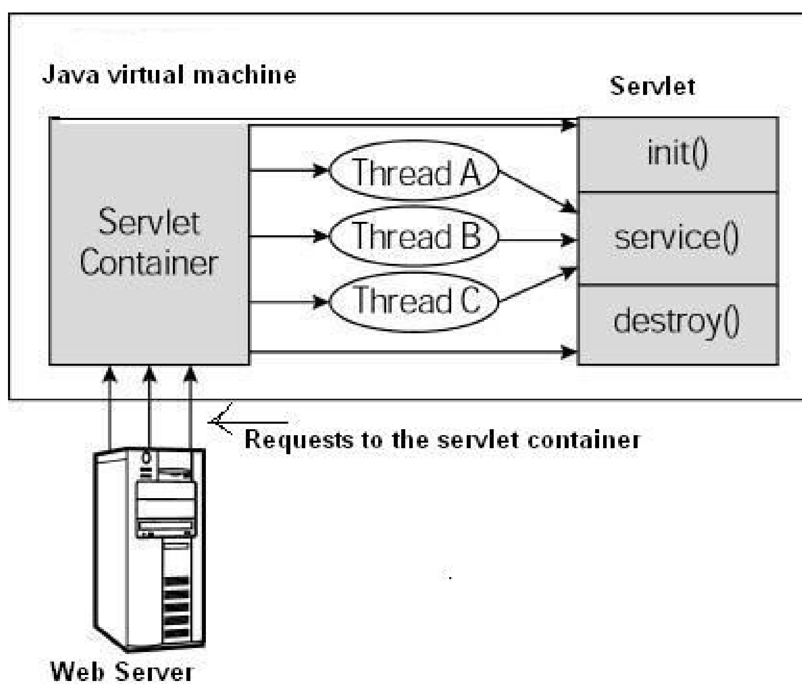
在调用 destroy() 方法之后，servlet 对象被标记为垃圾回收。destroy 方法定义如下所示：

```
  public void destroy() {
    // 终止化代码...
  }
```

架构图

下图显示了一个典型的 Servlet 生命周期方案。

- 第一个到达服务器的 HTTP 请求被委派到 Servlet 容器。
- Servlet 容器在调用 service() 方法之前加载 Servlet。
- 然后 Servlet 容器处理由多个线程产生的多个请求，每个线程执行一个单一的 Servlet 实例的 service() 方法。



Servlet 实例

Servlet 是服务 HTTP 请求并实现 **javax.servlet.Servlet** 接口的 Java 类。Web 应用程序开发人员通常编写 Servlet 来扩展 javax.servlet.http.HttpServlet，并实现 Servlet 接口的抽象类专门用来处理 HTTP 请求。

Hello World 示例代码

下面是 Servlet 输出 Hello World 的示例源代码：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloWorld extends HttpServlet {

  private String message;

  public void init() throws ServletException
  {
      // 执行必需的初始化
      message = "Hello World";
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      // 实际的逻辑是在这里
      PrintWriter out = response.getWriter();
      out.println("<h1>" + message + "</h1>");
  }

  public void destroy()
  {
      // 什么也不做
  }
}
```

编译 Servlet

让我们把上面的代码写在 HelloWorld.java 文件中，把这个文件放在 C:\ServletDevel（在 Windows 上）或 /usr/ServletDevel（在 UNIX 上）中，您还需要把这些目录添加到 CLASSPATH 中。

假设您的环境已经正确地设置，进入 **ServletDevel** 目录，并编译 HelloWorld.java，如下所示：

```
$ javac HelloWorld.java
```

如果 Servlet 依赖于任何其他库，您必须在 CLASSPATH 中包含那些 JAR 文件。在这里，我只包含了 servlet-api.jar JAR 文件，因为我没有在 Hello World 程序中使用任何其他库。

该命令行使用 Sun Microsystems Java 软件开发工具包（JDK）内置的 javac 编译器。为使该命令正常工作，您必须 PATH 环境变量中使用的 Java SDK 的位置。

如果一切顺利，上面编译会在同一目录下生成 HelloWorld.class 文件。下一节将讲解已编译的 Servlet 如何部署在生产中。

Servlet 部署

默认情况下，Servlet 应用程序位于路径 <Tomcat-installation-directory>/webapps/ROOT 下，且类文件放在 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

如果您有一个完全合格的类名称 **com.myorg.MyServlet**，那么这个 Servlet 类必须位于 WEB-INF/classes/com/myorg/MyServlet.class 中。

现在，让我们把 HelloWorld.class 复制到 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中，并在位于 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ 的 **web.xml** 文件中创建以下条目：

```
    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/HelloWorld</url-pattern>
    </servlet-mapping>
```

上面的条目要被创建在 web.xml 文件中的 <web-app>...</web-app> 标签内。在该文件中可能已经有各种可用的条目，但不要在意。

到这里，您基本上已经完成了，现在让我们使用 <Tomcat-installation-directory>\bin\startup.bat（在 Windows 上）或 <Tomcat-installation-directory>/bin/startup.sh（在 Linux/Solaris 等上）启动 tomcat 服务器，最后在浏览器的地址栏中输入 **http://localhost:8080/HelloWorld**。如果一切顺利，您会看到下面的结果：


http://localhost:8080/HelloWorld - Windows Internet Explorer
http://localhost:8080/HelloWorld
File Edit View Favorites Tools Help
http://localhost:8080/HelloWorld
Hello World

Servlet 表单数据

很多情况下，需要传递一些信息，从浏览器到 Web 服务器，最终到后台程序。浏览器使用两种方法可将这些信息传递到 Web 服务器，分别为 GET 方法和 POST 方法。

GET 方法

GET 方法向页面请求发送已编码的用户信息。页面和已编码的信息中间用 ? 字符分隔，如下所示：

```
http://www.test.com/hello?key1=value1&key2=value2
```

GET 方法是默认的从浏览器向 Web 服务器传递信息的方法，它会产生一个很长的字符串，出现在浏览器的地址栏中。如果您要向服务器传递的是密码或其他的敏感信息，请不要使用 GET 方法。GET 方法有大小限制：请求字符串中最多只能有 1024 个字符。

这些信息使用 QUERY_STRING 头传递，并可以通过 QUERY_STRING 环境变量访问，Servlet 使用 **doGet()** 方法处理这种类型的请求。

POST 方法

另一个向后台程序传递信息的比较可靠的方法是 POST 方法。POST 方法打包信息的方式与 GET 方法基本相同，但是 POST 方法不是把信息作为 URL 中 ? 字符后的文本字符串进行发送，而是把这些信息作为一个单独的消息。消息以标准输出的形式传到后台程序，您可以解析和使用这些标准输出。Servlet 使用 doPost() 方法处理这种类型的请求。

使用 Servlet 读取表单数据

Servlet 处理表单数据，这些数据会根据不同的情况使用不同的方法自动解析：

- **getParameter()**：您可以调用 request.getParameter() 方法来获取表单参数的值。
- **getParameterValues()**：如果参数出现一次以上，则调用该方法，并返回多个值，例如复选框。
- **getParameterNames()**：如果您想要得到当前请求中的所有参数的完整列表，则调用该方法。

使用 URL 的 GET 方法实例

下面是一个简单的 URL，将使用 GET 方法向 HelloForm 程序传递两个值。

http://localhost:8080/HelloForm?first_name=ZARA&last_name=ALI

下面是处理 Web 浏览器输入的 **HelloForm.java** Servlet 程序。我们将使用 **getParameter()** 方法，可以很容易地访问传递的信息：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloForm extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "使用 GET 方法读取表单数据";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>名字</b>："
                + request.getParameter("first_name") + "\n" +
                "  <li><b>姓氏</b>："
                + request.getParameter("last_name") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
}
```

假设您的环境已经正确地设置，编译 HelloForm.java，如下所示：

```
$ javac HelloForm.java
```

如果一切顺利，上述编译会产生 **HelloForm.class** 文件。接下来，您就必须把该类文件复制到 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中，并在位于 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ 的 **web.xml** 文件中创建以下条目：

```
    <servlet>
        <servlet-name>HelloForm</servlet-name>
        <servlet-class>HelloForm</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloForm</servlet-name>
        <url-pattern>/HelloForm</url-pattern>
    </servlet-mapping>
```

现在在浏览器的地址栏中输入 *http://localhost:8080/HelloForm?first_name=ZARA&last_name=ALI*，并在触发上述命令之前确保已经启动 Tomcat 服务器。如果一切顺利，您会得到下面的结果：

```
<h1>使用 GET 方法读取表单数据</h1>

<ul>
<li><b>名字<b>：ZARA</li>
<li><b>姓氏<b>：ALI</li>
</ul>
```

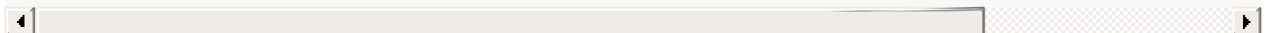
使用表单的 GET 方法实例

下面是一个简单的实例，使用 HTML 表单和提交按钮传递两个值。我们将使用相同的 Servlet HelloForm 来处理输入。

```
<html>
<body>
<form action="HelloForm" method="GET">
名字：<input type="text" name="first_name">
<br />
姓氏：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

保存这个 HTML 到 hello.htm 文件中，并把它放在 <Tomcat-installation-directory>/webapps/ROOT 目录下。当您访问 *http://localhost:8080/Hello.htm* 时，下面是上面表单的实际输出。

```
<form action="javascript:void();" method="get" target="_blank">名字：<input type="text" na
姓氏：<input type="text" name="last_name"> <input type="button" value="提交"></form>
```



尝试输入名字和姓氏，然后点击"提交"按钮，在您本机上查看输出结果。基于所提供的输入，它会产生与上一个实例类似的结果。

使用表单的 POST 方法实例

让我们对上面的 Servlet 做小小的修改，以便它可以处理 GET 和 POST 方法。下面的 **HelloForm.java** Servlet 程序使用 GET 和 POST 方法处理由 Web 浏览器给出的输入。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloForm extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Using GET Method to Read Form Data";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>名字</b>："
                + request.getParameter("first_name") + "\n" +
                "  <li><b>姓氏</b>："
                + request.getParameter("last_name") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，编译部署上述的 Servlet，并使用带有 POST 方法的 Hello.htm 进行测试，如下所示：

```
<html>
<body>
<form action="HelloForm" method="POST">
名字：<input type="text" name="first_name">
<br />
姓氏：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

下面是上面表单的实际输出，尝试输入名字和姓氏，然后点击"提交"按钮，在您本机上查看输出结果。

```
<form action="javascript:void();" method="get" target="_blank">名字：<input type="text" na
姓氏：<input type="text" name="last_name"> <input type="button" value="提交"></form>
```

基于所提供的输入，它会产生与上一个实例类似的结果。

将复选框数据传递到 Servlet 程序

当需要选择一个以上的选项时，则使用复选框。

下面是一个 HTML 代码实例 CheckBox.htm，一个带有两个复选框的表单。

```
<html>
<body>
<form action="CheckBox" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> 数学
<input type="checkbox" name="physics"  /> 物理
<input type="checkbox" name="chemistry" checked="checked" />
                                                   化学
<input type="submit" value="选择学科" />
</form>
</body>
</html>
```

这段代码的结果是下面的表单：

```
<form action="javascript:void();" method="get" target="_blank"><input type="checkbox" nam
```

下面是 CheckBox.java Servlet 程序，处理 Web 浏览器给出的复选框输入。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class CheckBox extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "读取复选框数据";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>数学标识：</b>: "
                + request.getParameter("maths") + "\n" +
                "  <li><b>物理标识：</b>: "
                + request.getParameter("physics") + "\n" +
                "  <li><b>化学标识：</b>: "
                + request.getParameter("chemistry") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

上面的实例将显示下面的结果：

```
<h1>读取复选框数据</h1>

<ul>
<li><b>数学标识：</b>on</li>
<li><b>物理标识：</b>null</li>
<li><b>化学标识：</b>on</li>
</ul>
```

读取所有的表单参数

以下是通用的实例，使用 HttpServletRequest 的 **getParameterNames()** 方法读取所有可用的表单参数。该方法返回一个枚举，其中包含未指定顺序的参数名。

一旦我们有一个枚举，我们可以以标准方式循环枚举，使用 *hasMoreElements()* 方法来确定何时停止，使用 *nextElement()* 方法来获取每个参数的名称。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class ReadParams extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "读取所有的表单数据";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<table width=\"100%\" border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        "<th>参数名称</th><th>参数值</th>\n"+
        "</tr>\n");

      Enumeration paramNames = request.getParameterNames();

      while(paramNames.hasMoreElements()) {
         String paramName = (String)paramNames.nextElement();
         out.print("<tr><td>" + paramName + "</td>\n<td>");
         String[] paramValues =
                request.getParameterValues(paramName);
         // 读取单个值的数据
         if (paramValues.length == 1) {
           String paramValue = paramValues[0];
           if (paramValue.length() == 0)
             out.println("<i>No Value</i>");
           else
             out.println(paramValue);
         } else {
             // 读取多个值的数据
             out.println("<ul>");
             for(int i=0; i < paramValues.length; i++) {
                out.println("<li>" + paramValues[i]);
             }
             out.println("</ul>");
         }
      }
      out.println("</tr>\n</table>\n</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，通过下面的表单尝试上面的 Servlet：

```
<html>
<body>
<form action="ReadParams" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> 数学
<input type="checkbox" name="physics"  /> 物理
<input type="checkbox" name="chemistry" checked="checked" /> 化学
<input type="submit" value="选择学科" />
</form>
</body>
</html>
```

现在使用上面的表单调用 Servlet，将产生以下结果：

读取所有的表单数据

参数名称	参数值
maths	on
chemistry	on

您可以尝试使用上面的 Servlet 来读取其他的表单数据，比如文本框、单选按钮或下拉框等。

Servlet 客户端 HTTP 请求

当浏览器请求网页时，它会向 Web 服务器发送特定信息，这些信息不能被直接读取，因为这些信息是作为 HTTP 请求的头的一部分进行传输的。您可以查看 HTTP 协议 了解更多相关信息。

以下是来自于浏览器端的重要头信息，您可以在 Web 编程中频繁使用：

头信息	描述
Accept	这个头信息指定浏览器或其他客户端可以处理的 MIME 类型。值 **image/png** 或 **image/jpeg** 是最常见的两种可能值。
Accept-Charset	这个头信息指定浏览器可以用来显示信息的字符集。例如 ISO-8859-1。
Accept-Encoding	这个头信息指定浏览器知道如何处理的编码类型。值 **gzip** 或 **compress** 是最常见的两种可能值。
Accept-Language	这个头信息指定客户端的首选语言，在这种情况下，Servlet 会产生多种语言的结果。例如，en、en-us、ru 等。
Authorization	这个头信息用于客户端在访问受密码保护的网页时识别自己的身份。
Connection	这个头信息指示客户端是否可以处理持久 HTTP 连接。持久连接允许客户端或其他浏览器通过单个请求来检索多个文件。值 **Keep-Alive** 意味着使用了持续连接。
Content-Length	这个头信息只适用于 POST 请求，并给出 POST 数据的大小（以字节为单位）。
Cookie	这个头信息把之前发送到浏览器的 cookies 返回到服务器。
Host	这个头信息指定原始的 URL 中的主机和端口。
If-Modified-Since	这个头信息表示只有当页面在指定的日期后已更改时，客户端想要的页面。如果没有新的结果可以使用，服务器会发送一个 304 代码，表示 **Not Modified** 头信息。
If-Unmodified-Since	这个头信息是 If-Modified-Since 的对立面，它指定只有当文档早于指定日期时，操作才会成功。
Referer	这个头信息指示所指向的 Web 页的 URL。例如，如果您在网页 1，点击一个链接到网页 2，当浏览器请求网页 2 时，网页 1 的 URL 就会包含在 Referer 头信息中。
User-Agent	这个头信息识别发出请求的浏览器或其他客户端，并可以向不同类型的浏览器返回不同的内容。

读取 HTTP 头的方法

下面的方法可用在 Servlet 程序中读取 HTTP 头。这些方法通过 *HttpServletRequest* 对象可用。

方法	描述
Cookie[] getCookies()	返回一个数组，包含客户端发送该请求的所有的 Cookie 对象。
Enumeration getAttributeNames()	返回一个枚举，包含提供给该请求可用的属性名称。
Enumeration getHeaderNames()	返回一个枚举，包含在该请求中包含的所有的头名。
Enumeration getParameterNames()	返回一个 String 对象的枚举，包含在该请求中包含的参数的名称。
HttpSession getSession()	返回与该请求关联的当前 session 会话，或者如果请求没有 session 会话，则创建一个。
HttpSession getSession(boolean create)	返回与该请求关联的当前 HttpSession，或者如果没有当前会话，且创建是真的，则返回一个新的 session 会话。
Locale getLocale()	基于 Accept-Language 头，返回客户端接受内容的首选的区域设置。
Object getAttribute(String name)	以对象形式返回已命名属性的值，如果没有给定名称的属性存在，则返回 null。
ServletInputStream getInputStream()	使用 ServletInputStream，以二进制数据形式检索请求的主体。
String getAuthType()	返回用于保护 Servlet 的身份验证方案的名称，例如，"BASIC" 或 "SSL"，如果JSP没有受到保护则返回 null。
String getCharacterEncoding()	返回请求主体中使用的字符编码的名称。
String getContentType()	返回请求主体的 MIME 类型，如果不知道类型则返回 null。
String getContextPath()	返回指示请求上下文的请求 URI 部分。
String getHeader(String name)	以字符串形式返回指定的请求头的值。
String getMethod()	返回请求的 HTTP 方法的名称，例如，GET、POST 或 PUT。
String getParameter(String name)	以字符串形式返回请求参数的值，或者如果参数不存在则返回 null。
String getPathInfo()	当请求发出时，返回与客户端发送的 URL 相关的任何额外的路径信息。
String getProtocol()	返回请求协议的名称和版本。

String getQueryString()	返回包含在路径后的请求 URL 中的查询字符串。
String getRemoteAddr()	返回发送请求的客户端的互联网协议（IP）地址。
String getRemoteHost()	返回发送请求的客户端的完全限定名称。
String getRemoteUser()	如果用户已通过身份验证，则返回发出请求的登录用户，或者如果用户未通过身份验证，则返回 null。
String getRequestURI()	从协议名称直到 HTTP 请求的第一行的查询字符串中，返回该请求的 URL 的一部分。
String getRequestedSessionId()	返回由客户端指定的 session 会话 ID。
String getServletPath()	返回调用 JSP 的请求的 URL 的一部分。
String[] getParameterValues(String name)	返回一个字符串对象的数组，包含所有给定的请求参数的值，如果参数不存在则返回 null。
boolean isSecure()	返回一个布尔值，指示请求是否使用安全通道，如 HTTPS。
int getContentLength()	以字节为单位返回请求主体的长度，并提供输入流，或者如果长度未知则返回 -1。
int getIntHeader(String name)	返回指定的请求头的值为一个 int 值。
int getServerPort()	返回接收到这个请求的端口号。

HTTP Header 请求实例

下面的实例使用 HttpServletRequest 的 **getHeaderNames()** 方法读取 HTTP 头信息。该方法返回一个枚举，包含与当前的 HTTP 请求相关的头信息。

一旦我们有一个枚举，我们可以以标准方式循环枚举，使用 *hasMoreElements()* 方法来确定何时停止，使用 *nextElement()* 方法来获取每个参数的名称。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class DisplayHeader extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "HTTP Header 请求实例";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n"+
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<table width=\"100%\" border=\"1\" align=\"center\">\n" +
        "<tr bgcolor=\"#949494\">\n" +
        "<th>Header 名称</th><th>Header 值</th>\n"+
        "</tr>\n");

      Enumeration headerNames = request.getHeaderNames();

      while(headerNames.hasMoreElements()) {
         String paramName = (String)headerNames.nextElement();
         out.print("<tr><td>" + paramName + "</td>\n");
         String paramValue = request.getHeader(paramName);
         out.println("<td> " + paramValue + "</td></tr>\n");
      }
      out.println("</table>\n</body></html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，调用上面的 Servlet 会产生以下结果：

```
<h1>HTTP Header 请求实例</h1>

<table>

<tbody>

<tr bgcolor="#949494"><th>Header 名称</th><th>Header 值</th></tr>

<tr><td>accept</td><td>*/*</td></tr>

<tr><td>accept-language</td><td>en-us</td></tr>

<tr><td>user-agent</td><td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0

<tr><td>accept-encoding</td><td>gzip, deflate</td></tr>

<tr><td>host</td><td>localhost:8080</td></tr>

<tr><td>connection</td><td>Keep-Alive</td></tr>

<tr><td>cache-control</td><td>no-cache</td></tr>

</tbody>

</table>
```

Servlet 服务器 HTTP 响应

正如前面的章节中讨论的那样，当一个 Web 服务器响应一个 HTTP 请求时，响应通常包括一个状态行、一些响应报头、一个空行和文档。一个典型的响应如下所示：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包括 HTTP 版本（在本例中为 HTTP/1.1）、一个状态码（在本例中为 200）和一个对应于状态码的短消息（在本例中为 OK）。

下表总结了从 Web 服务器端返回到浏览器的最有用的 HTTP 1.1 响应报头，您会在 Web 编程中频繁地使用它们：

头信息	描述
Allow	这个头信息指定服务器支持的请求方法（GET、POST 等）。
Cache-Control	这个头信息指定响应文档在何种情况下可以安全地缓存。可能的值有：**public**、**private** 或 **no-cache** 等。Public 意味着文档是可缓存，Private 意味着文档是单个用户私用文档，且只能存储在私有（非共享）缓存中，no-cache 意味着文档不应被缓存。
Connection	这个头信息指示浏览器是否使用持久 HTTP 连接。值 **close** 指示浏览器不使用持久 HTTP 连接，值 **keep-alive** 意味着使用持久连接。
Content-Disposition	这个头信息可以让您请求浏览器要求用户以给定名称的文件把响应保存到磁盘。
Content-Encoding	在传输过程中，这个头信息指定页面的编码方式。
Content-Language	这个头信息表示文档编写所使用的语言。例如，en、en-us、ru 等。
Content-Length	这个头信息指示响应中的字节数。只有当浏览器使用持久（keep-alive）HTTP 连接时才需要这些信息。
Content-Type	这个头信息提供了响应文档的 MIME（Multipurpose Internet Mail Extension）类型。
Expires	这个头信息指定内容过期的时间，在这之后内容不再被缓存。
Last-Modified	这个头信息指示文档的最后修改时间。然后，客户端可以缓存文件，并在以后的请求中通过 **If-Modified-Since** 请求头信息提供一个日期。
Location	这个头信息应被包含在所有的带有状态码的响应中。在 300s 内，这会通知浏览器文档的地址。浏览器会自动重新连接到这个位置，并获取新的文档。
Refresh	这个头信息指定浏览器应该如何尽快请求更新的页面。您可以指定页面刷新的秒数。
Retry-After	这个头信息可以与 503（Service Unavailable 服务不可用）响应配合使用，这会告诉客户端多久就可以重复它的请求。
Set-Cookie	个头信息指定一个与页面关联的 cookie。

设置 HTTP 响应报头的方法

下面的方法可用于在 Servlet 程序中设置 HTTP 响应报头。这些方法通过 *HttpServletResponse* 对象可用。

方法	描述
String encodeRedirectURL(String url)	为 sendRedirect 方法中使用的指定的 URL 进行编码，或者如果编码不是必需的，则返回 URL 未改变。

String encodeURL(String url)	对包含 session 会话 ID 的指定 URL 进行编码，或者如果编码不是必需的，则返回 URL 未改变。
boolean containsHeader(String name)	返回一个布尔值，指示是否已经设置已命名的响应报头。
boolean isCommitted()	返回一个布尔值，指示响应是否已经提交。
void addCookie(Cookie cookie)	把指定的 cookie 添加到响应。
void addDateHeader(String name, long date)	添加一个带有给定的名称和日期值的响应报头。
void addHeader(String name, String value)	添加一个带有给定的名称和值的响应报头。
void addIntHeader(String name, int value)	添加一个带有给定的名称和整数值的响应报头。
void flushBuffer()	强制任何在缓冲区中的内容被写入到客户端。
void reset()	清除缓冲区中存在的任何数据，包括状态码和头。
void resetBuffer()	清除响应中基础缓冲区的内容，不清除状态码和头。
void sendError(int sc)	使用指定的状态码发送错误响应到客户端，并清除缓冲区。
void sendError(int sc, String msg)	使用指定的状态发送错误响应到客户端。
void sendRedirect(String location)	使用指定的重定向位置 URL 发送临时重定向响应到客户端。
void setBufferSize(int size)	为响应主体设置首选的缓冲区大小。
void setCharacterEncoding(String charset)	设置被发送到客户端的响应的字符编码（MIME 字符集）例如，UTF-8。
void setContentLength(int len)	设置在 HTTP Servlet 响应中的内容主体的长度，该方法设置 HTTP Content-Length 头。
void setContentType(String type)	如果响应还未被提交，设置被发送到客户端的响应的内容类型。
void setDateHeader(String name, long date)	设置一个带有给定的名称和日期值的响应报头。
void setHeader(String name, String value)	设置一个带有给定的名称和值的响应报头。
void setIntHeader(String name, int value)	设置一个带有给定的名称和整数值的响应报头。
void setLocale(Locale loc)	如果响应还未被提交，设置响应的区域。
void setStatus(int sc)	为该响应设置状态码。

HTTP Header 响应实例

您已经在前面的实例中看到 setContentType() 方法，下面的实例也使用了同样的方法，此外，我们会用 **setIntHeader()** 方法来设置 **Refresh** 头。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class Refresh extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置刷新自动加载时间为 5 秒
      response.setIntHeader("Refresh", 5);

      // 设置响应内容类型
      response.setContentType("text/html");

      // Get current time
      Calendar calendar = new GregorianCalendar();
      String am_pm;
      int hour = calendar.get(Calendar.HOUR);
      int minute = calendar.get(Calendar.MINUTE);
      int second = calendar.get(Calendar.SECOND);
      if(calendar.get(Calendar.AM_PM) == 0)
        am_pm = "AM";
      else
        am_pm = "PM";

      String CT = hour+":"+ minute +":"+ second +" "+ am_pm;

      PrintWriter out = response.getWriter();
      String title = "自动刷新 Header 设置";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n"+
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<p>当前时间是：" + CT + "</p>\n");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，调用上面的 Servlet，每隔 5 秒会显示当前系统时间。只要运行 Servlet 并稍等片刻，即可看到如下的结果：

```
<h1>自动刷新 Header 设置</h1>

当前时间是：9:44:50 PM
```

Servlet HTTP 状态码

HTTP 请求和 HTTP 响应消息的格式是类似的，结构如下：

- 初始状态行 + 回车换行符（回车+换行）
- 零个或多个标题行+回车换行符
- 一个空白行，即回车换行符
- 一个可选的消息主体，比如文件、查询数据或查询输出

例如，服务器的响应头如下所示：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包括 HTTP 版本（在本例中为 HTTP/1.1）、一个状态码（在本例中为 200）和一个对应于状态码的短消息（在本例中为 OK）。

以下是可能从 Web 服务器返回的 HTTP 状态码和相关的信息列表：

代码	消息	描述
100	Continue	只有请求的一部分已经被服务器接收，但只要它没有被拒绝，客户端应继续该请求。
101	Switching Protocols	服务器切换协议。
200	OK	请求成功。
201	Created	该请求是完整的，并创建一个新的资源。
202	Accepted	该请求被接受处理，但是该处理是不完整的。
203	Non-authoritative Information	
204	No Content	
205	Reset Content	

206	Partial Content	
300	Multiple Choices	链接列表。用户可以选择一个链接，进入到该位置。最多五个地址。
301	Moved Permanently	所请求的页面已经转移到一个新的 URL。
302	Found	所请求的页面已经临时转移到一个新的 URL。
303	See Other	所请求的页面可以在另一个不同的 URL 下被找到。
304	Not Modified	
305	Use Proxy	
306	*Unused*	在以前的版本中使用该代码。现在已不再使用它，但代码仍被保留。
307	Temporary Redirect	所请求的页面已经临时转移到一个新的 URL。
400	Bad Request	服务器不理解请求。
401	Unauthorized	所请求的页面需要用户名和密码。
402	Payment Required	您还不能使用该代码。
403	Forbidden	禁止访问所请求的页面。
404	Not Found	服务器无法找到所请求的页面。.
405	Method Not Allowed	在请求中指定的方法是不允许的。
406	Not Acceptable	服务器只生成一个不被客户端接受的响应。
407	Proxy Authentication Required	在请求送达之前，您必须使用代理服务器的验证。
408	Request Timeout	请求需要的时间比服务器能够等待的时间长，超时。
409	Conflict	请求因为冲突无法完成。
410	Gone	所请求的页面不再可用。
411	Length Required	"Content-Length" 未定义。服务器无法处理客户端发送的不带 Content-Length 的请求信息。
412	Precondition Failed	请求中给出的先决条件被服务器评估为 false。
413	Request Entity Too Large	服务器不接受该请求，因为请求实体过大。
414	Request-url Too Long	服务器不接受该请求，因为 URL 太长。当您转换一个 "post" 请求为一个带有长的查询信息的 "get" 请求时发生。

415	Unsupported Media Type	服务器不接受该请求，因为媒体类型不被支持。
417	Expectation Failed	
500	Internal Server Error	未完成的请求。服务器遇到了一个意外的情况。
501	Not Implemented	未完成的请求。服务器不支持所需的功能。
502	Bad Gateway	未完成的请求。服务器从上游服务器收到无效响应。
503	Service Unavailable	未完成的请求。服务器暂时超载或死机。
504	Gateway Timeout	网关超时。
505	HTTP Version Not Supported	服务器不支持"HTTP协议"版本。

设置 HTTP 状态代码的方法

下面的方法可用于在 Servlet 程序中设置 HTTP 状态码。这些方法通过 *HttpServletResponse* 对象可用。

方法	描述
public void setStatus (int statusCode)	该方法设置一个任意的状态码。setStatus 方法接受一个 int（状态码）作为参数。如果您的反应包含了一个特殊的状态码和文档，请确保在使用 *PrintWriter* 实际返回任何内容之前调用 setStatus。
public void sendRedirect(String url)	该方法生成一个 302 响应，连同一个带有新文档 URL 的 *Location* 头。
public void sendError(int code, String message)	该方法发送一个状态码（通常为 404），连同一个在 HTML 文档内部自动格式化并发送到客户端的短消息。

HTTP 状态码实例

下面的例子把 407 错误代码发送到客户端浏览器，浏览器会显示 "Need authentication!!!" 消息。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class showError extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置错误代码和原因
      response.sendError(407, "Need authentication!!!" );
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在，调用上面的 Servlet 将显示以下结果：

```
<h2>HTTP Status 407 - Need authentication!!!</h2>
<b>type</b> Status report
<b>message</b> <u>Need authentication!!!</u>
<b>description</b> <u>The client must first authenticate itself with the proxy (Need auth
<h3>Apache Tomcat/5.5.29</h3>
```

Servlet 编写过滤器

Servlet 过滤器是可用于 Servlet 编程的 Java 类，有以下目的：

- 在客户端的请求访问后端资源之前，拦截这些请求。
- 在服务器的响应发送回客户端之前，处理这些响应。

根据规范建议的各种类型的过滤器：

- 身份验证过滤器（Authentication Filters）。
- 数据压缩过滤器（Data compression Filters）。
- 加密过滤器（Encryption Filters）。
- 触发资源访问事件过滤器。
- 图像转换过滤器（Image Conversion Filters）。
- 日志记录和审核过滤器（Logging and Auditing Filters）。
- MIME-TYPE 链过滤器（MIME-TYPE Chain Filters）。
- 标记化过滤器（Tokenizing Filters）。
- XSL/T 过滤器（XSL/T Filters），转换 XML 内容。

过滤器被部署在部署描述符文件 **web.xml** 中，然后映射到您的应用程序的部署描述符中的 Servlet 名称或 URL 模式。

当 Web 容器启动 Web 应用程序时，它会为您在部署描述符中声明的每一个过滤器创建一个实例。该过滤器执行的顺序是按它们在部署描述符中声明的顺序。

Servlet 过滤器方法

过滤器是一个实现了 javax.servlet.Filter 接口的 Java 类。javax.servlet.Filter 接口定义了三个方法：

方法	描述
public void doFilter (ServletRequest, ServletResponse, FilterChain)	该方法在每次一个请求/响应对因客户端在链的末端请求资源而通过链传递时由容器调用。
public void init(FilterConfig filterConfig)	该方法由 Web 容器调用，指示一个过滤器被放入服务。
public void destroy()	该方法由 Web 容器调用，指示一个过滤器被取出服务。

Servlet 过滤器实例

以下是 Servlet 过滤器的实例，将输出客户端的 IP 地址和当前的日期时间。本实例让您对 Servlet 过滤器有基本的了解，您可以使用相同的概念编写更复杂的过滤器应用程序：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 实现 Filter 类
public class LogFilter implements Filter  {
   public void  init(FilterConfig config)
                         throws ServletException{
      // 获取初始化参数
      String testParam = config.getInitParameter("test-param");

      // 输出初始化参数
      System.out.println("Test Param: " + testParam);
   }
   public void  doFilter(ServletRequest request,
                 ServletResponse response,
                 FilterChain chain)
                 throws java.io.IOException, ServletException {

      // 获取客户机的 IP 地址
      String ipAddress = request.getRemoteAddr();

      // 记录 IP 地址和当前时间戳
      System.out.println("IP "+ ipAddress + ", Time "
                                 + new Date().toString());

      // 把请求传回过滤链
      chain.doFilter(request,response);
   }
   public void destroy( ){
      /* 在 Filter 实例被 Web 容器从服务移除之前调用 */
   }
}
```

以通常的方式编译 **LogFilter.java**，把您的类文件放入 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

Web.xml 中的 Servlet 过滤器映射（Servlet Filter Mapping）

定义过滤器，然后映射到一个 URL 或 Servlet，这与定义 Servlet，然后映射到一个 URL 模式方式大致相同。在部署描述符文件 **web.xml** 中为 filter 标签创建下面的条目：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

上述过滤器适用于所有的 Servlet，因为我们在配置中指定 */* 。如果您只想在少数的 Servlet 上应用过滤器，您可以指定一个特定的 Servlet 路径。

现在试着以常用的方式调用任何 Servlet，您将会看到在 Web 服务器中生成的日志。您也可以使用 Log4J 记录器来把上面的日志记录到一个单独的文件中。

使用多个过滤器

Web 应用程序可以根据特定的目的定义若干个不同的过滤器。假设您定义了两个过滤器 *AuthenFilter* 和 *LogFilter*。您需要创建一个如下所述的不同的映射，其余的处理与上述所讲解的大致相同：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>AuthenFilter</filter-name>
   <filter-class>AuthenFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

过滤器的应用顺序

web.xml 中的 filter-mapping 元素的顺序决定了 Web 容器应用过滤器到 Servlet 的顺序。若要反转过滤器的顺序，您只需要在 web.xml 文件中反转 filter-mapping 元素即可。

例如，上面的实例将先应用 LogFilter，然后再应用 AuthenFilter，但是下面的实例将颠倒这个顺序：

```
<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Servlet 异常处理

当一个 Servlet 抛出一个异常时，Web 容器在使用了 exception-type 元素的 **web.xml** 中搜索与抛出异常类型相匹配的配置。

您必须在 web.xml 中使用 **error-page** 元素来指定对特定异常 或 HTTP 状态码 作出相应的 Servlet 调用。

web.xml 配置

假设，有一个 *ErrorHandler* 的 Servelt 在任何已定义的异常或错误出现时被调用。以下将是在 web.xml 中创建的项。

```
<!-- servlet 定义 -->
<servlet>
        <servlet-name>ErrorHandler</servlet-name>
        <servlet-class>ErrorHandler</servlet-class>
</servlet>
<!-- servlet 映射 -->
<servlet-mapping>
        <servlet-name>ErrorHandler</servlet-name>
        <url-pattern>/ErrorHandler</url-pattern>
</servlet-mapping>

<!-- error-code 相关的错误页面 -->
<error-page>
    <error-code>404</error-code>
    <location>/ErrorHandler</location>
</error-page>
<error-page>
    <error-code>403</error-code>
    <location>/ErrorHandler</location>
</error-page>

<!-- exception-type 相关的错误页面 -->
<error-page>
    <exception-type>
          javax.servlet.ServletException
    </exception-type >
    <location>/ErrorHandler</location>
</error-page>

<error-page>
    <exception-type>java.io.IOException</exception-type >
    <location>/ErrorHandler</location>
</error-page>
```

如果您想对所有的异常有一个通用的错误处理程序，那么应该定义下面的 error-page，而不是为每个异常定义单独的 error-page 元素：

```
<error-page>
    <exception-type>java.lang.Throwable</exception-type >
    <location>/ErrorHandler</location>
</error-page>
```

以下是关于上面的 web.xml 异常处理要注意的点：

- Servelt ErrorHandler 与其他的 Servelt 的定义方式一样，且在 web.xml 中进行配置。
- 如果有错误状态代码出现，不管为 404（Not Found 未找到）或 403（Forbidden 禁止），则会调用 ErrorHandler 的 Servlet。
- 如果 Web 应用程序抛出 *ServletException* 或 *IOException*，那么 Web 容器会调用 ErrorHandler 的 Servlet。
- 您可以定义不同的错误处理程序来处理不同类型的错误或异常。上面的实例是非常通用的，希望您能通过实例理解基本的概念。

请求属性 - 错误/异常

以下是错误处理的 Servlet 可以访问的请求属性列表，用来分析错误/异常的性质。

属性	描述
javax.servlet.error.status_code	该属性给出状态码，状态码可被存储，并在存储为 java.lang.Integer 数据类型后可被分析。
javax.servlet.error.exception_type	该属性给出异常类型的信息，异常类型可被存储，并在存储为 java.lang.Class 数据类型后可被分析。
javax.servlet.error.message	该属性给出确切错误消息的信息，信息可被存储，并在存储为 java.lang.String 数据类型后可被分析。
javax.servlet.error.request_uri	该属性给出有关 URL 调用 Servlet 的信息，信息可被存储，并在存储为 java.lang.String 数据类型后可被分析。
javax.servlet.error.exception	该属性给出异常产生的信息，信息可被存储，并在存储为 java.lang.Throwable 数据类型后可被分析。
javax.servlet.error.servlet_name	该属性给出 Servlet 的名称，名称可被存储，并在存储为 java.lang.String 数据类型后可被分析。

Servlet 错误处理程序实例

以下是 Servlet 实例，将应对任何您所定义的错误或异常发生时的错误处理程序。

本实例让您对 Servlet 中的异常处理有基本的了解，您可以使用相同的概念编写更复杂的异常处理应用程序：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class ErrorHandler extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 分析 Servlet 异常
      Throwable throwable = (Throwable)
      request.getAttribute("javax.servlet.error.exception");
      Integer statusCode = (Integer)
      request.getAttribute("javax.servlet.error.status_code");
      String servletName = (String)
      request.getAttribute("javax.servlet.error.servlet_name");
      if (servletName == null){
         servletName = "Unknown";
      }
      String requestUri = (String)
      request.getAttribute("javax.servlet.error.request_uri");
      if (requestUri == null){
         requestUri = "Unknown";
      }

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Error/Exception Information";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n");

      if (throwable == null && statusCode == null){
         out.println("<h2>Error information is missing</h2>");
         out.println("Please return to the <a href=\"" +
           response.encodeURL("http://localhost:8080/") +
           "\">Home Page</a>.");
      }else if (statusCode != null){
         out.println("The status code : " + statusCode);
      }else{
         out.println("<h2 class="tutheader">Error information</h2>");
         out.println("Servlet Name : " + servletName +
                             "</br></br>");
         out.println("Exception Type : " +
                             throwable.getClass( ).getName( ) +
                             "</br></br>");
         out.println("The request URI: " + requestUri +
                             "<br><br>");
         out.println("The exception message: " +
                                 throwable.getMessage( ));
      }
      out.println("</body>");
      out.println("</html>");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

以通常的方式编译 **ErrorHandler.java**，把您的类文件放入<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

让我们在 web.xml 文件中添加如下配置来处理异常：

```
<servlet>
        <servlet-name>ErrorHandler</servlet-name>
        <servlet-class>ErrorHandler</servlet-class>
</servlet>
<!-- servlet mappings -->
<servlet-mapping>
        <servlet-name>ErrorHandler</servlet-name>
        <url-pattern>/ErrorHandler</url-pattern>
</servlet-mapping>
<error-page>
    <error-code>404</error-code>
    <location>/ErrorHandler</location>
</error-page>
<error-page>
    <exception-type>java.lang.Throwable</exception-type >
    <location>/ErrorHandler</location>
</error-page>
```

现在，尝试使用一个会产生异常的 Servlet，或者输入一个错误的 URL，这将触发 Web 容器调用 **ErrorHandler** 的 Servlet，并显示适当的消息。例如，如果您输入了一个错误的 URL，那么它将显示下面的结果：

```
The status code : 404
```

上面的代码在某些 Web 浏览器中可能无法正常工作。因此，请尽量尝试使用 Mozilla 和 Safari 浏览器，在这两种浏览器中它们应该能够正常工作。

Servlet Cookies 处理

Cookies 是存储在客户端计算机上的文本文件，并保留了各种跟踪信息。Java Servlet 显然支持 HTTP Cookies。

识别返回用户包括三个步骤：

- 服务器脚本向浏览器发送一组 Cookies。例如：姓名、年龄或识别号码等。
- 浏览器将这些信息存储在本地计算机上，以备将来使用。
- 当下一次浏览器向 Web 服务器发送任何请求时，浏览器会把这些 Cookies 信息发送到服务器，服务器将使用这些信息来识别用户。

本章将向您讲解如何设置或重置 Cookies，如何访问它们，以及如何将它们删除。

Cookie 剖析

Cookies 通常设置在 HTTP 头信息中（虽然 JavaScript 也可以直接在浏览器上设置一个 Cookie）。设置 Cookie 的 Servlet 会发送如下的头信息：

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
                 path=/; domain=w3cschool.cc
Connection: close
Content-Type: text/html
```

正如您所看到的，Set-Cookie 头包含了一个名称值对、一个 GMT 日期、一个路径和一个域。名称和值会被 URL 编码。expires 字段是一个指令，告诉浏览器在给定的时间和日期之后"忘记"该 Cookie。

如果浏览器被配置为存储 Cookies，它将会保留此信息直到到期日期。如果用户的浏览器指向任何匹配该 Cookie 的路径和域的页面，它会重新发送 Cookie 到服务器。浏览器的头信息可能如下所示：

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

Servlet 就能够通过请求方法 *request.getCookies()* 访问 Cookie，该方法将返回一个 *Cookie* 对象的数组。

Servlet Cookies 方法

以下是在 Servlet 中操作 Cookies 时可使用的有用的方法列表。

方法	描述
public void setDomain(String pattern)	该方法设置 cookie 适用的域，例如 w3cschool.cc。
public String getDomain()	该方法获取 cookie 适用的域，例如 w3cschool.cc。
public void setMaxAge(int expiry)	该方法设置 cookie 过期的时间（以秒为单位）。如果不这样设置，cookie 只会在当前 session 会话中持续有效。
public int getMaxAge()	该方法返回 cookie 的最大生存周期（以秒为单位），默认情况下，-1 表示 cookie 将持续下去，直到浏览器关闭。
public String getName()	该方法返回 cookie 的名称。名称在创建后不能改变。
public void setValue(String newValue)	该方法设置与 cookie 关联的值。
public String getValue()	该方法获取与 cookie 关联的值。
public void setPath(String uri)	该方法设置 cookie 适用的路径。如果您不指定路径，与当前页面相同目录下的（包括子目录下的）所有 URL 都会返回 cookie。
public String getPath()	该方法获取 cookie 适用的路径。
public void setSecure(boolean flag)	该方法设置布尔值，表示 cookie 是否应该只在加密的（即 SSL）连接上发送。
public void setComment(String purpose)	该方法规定了描述 cookie 目的的注释。该注释在浏览器向用户呈现 cookie 时非常有用。
public String getComment()	该方法返回了描述 cookie 目的的注释，如果 cookie 没有注释则返回 null。

通过 Servlet 设置 Cookies

通过 Servlet 设置 Cookies 包括三个步骤：

(1) 创建一个 **Cookie** 对象：您可以调用带有 cookie 名称和 cookie 值的 Cookie 构造函数，cookie 名称和 cookie 值都是字符串。

```
Cookie cookie = new Cookie("key","value");
```

请记住，无论是名字还是值，都不应该包含空格或以下任何字符：

```
[ ] ( ) = , " / ? @ : ;
```

(2) 设置最大生存周期：您可以使用 setMaxAge 方法来指定 cookie 能够保持有效的时间（以秒为单位）。下面将设置一个最长有效期为 24 小时的 cookie。

```
cookie.setMaxAge(60*60*24);
```

(3) 发送 **Cookie** 到 **HTTP** 响应头：您可以使用 **response.addCookie** 来添加 HTTP 响应头中的 Cookies，如下所示：

```
response.addCookie(cookie);
```

实例

让我们修改我们的 表单数据实例，为名字和姓氏设置 Cookies。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class HelloForm extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 为名字和姓氏创建 Cookies
      Cookie firstName = new Cookie("first_name",
                      request.getParameter("first_name"));
      Cookie lastName = new Cookie("last_name",
                      request.getParameter("last_name"));

      // 为两个 Cookies 设置过期日期为 24 小时后
      firstName.setMaxAge(60*60*24);
      lastName.setMaxAge(60*60*24);

      // 在响应头中添加两个 Cookies
      response.addCookie( firstName );
      response.addCookie( lastName );

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "设置 Cookies 实例";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" +
                "<h1 align=\"center\">" + title + "</h1>\n" +
                "<ul>\n" +
                "  <li><b>名字</b>："
                + request.getParameter("first_name") + "\n" +
                "  <li><b>姓氏</b>："
                + request.getParameter("last_name") + "\n" +
                "</ul>\n" +
                "</body></html>");
  }
}
```

编译上面的 Servlet **HelloForm**，并在 web.xml 文件中创建适当的条目，最后尝试下面的 HTML 页面来调用 Servlet。

```
<html>
<body>
<form action="HelloForm" method="GET">
名字：<input type="text" name="first_name">
<br />
姓氏：<input type="text" name="last_name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

保存上面的 HTML 内容到文件 hello.htm 中，并把它放在 <Tomcat-installation-directory>/webapps/ROOT 目录中。当您访问 *http://localhost:8080/Hello.htm* 时，上面表单的实际输出如下所示：

名字：[ ] 姓氏：[ ] [提交]

尝试输入名字和姓氏，然后点击"提交"按钮，名字和姓氏将显示在屏幕上，同时会设置 firstName 和 lastName 这两个 Cookies，当下次您按下提交按钮时，会被这两个 Cookies 传回到服务器。

下一节会讲解如何在 Web 应用程序中访问这些 Cookies。

通过 Servlet 读取 Cookies

要读取 Cookies，您需要通过调用 *HttpServletRequest* 的 **getCookies( )** 方法创建一个 *javax.servlet.http.Cookie* 对象的数组。然后循环遍历数组，并使用 getName() 和 getValue() 方法来访问每个 cookie 和关联的值。

实例

让我们读取上面的实例中设置的 Cookies

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class ReadCookies extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      Cookie cookie = null;
      Cookie[] cookies = null;
      // 获取与该域相关的 Cookies 的数组
      cookies = request.getCookies();

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Reading Cookies Example";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" );
      if( cookies != null ){
         out.println("<h2>查找 Cookies 名称和值</h2>");
         for (int i = 0; i < cookies.length; i++){
            cookie = cookies[i];
            out.print("名称：" + cookie.getName( ) + ", ");
            out.print("值：" + cookie.getValue( )+" <br/>");
         }
      }else{
          out.println(
            "<h2 class="tutheader">未找到 Cookies</h2>");
      }
      out.println("</body>");
      out.println("</html>");
   }
}
```

编译上面的 Servlet **ReadCookies**，并在 web.xml 文件中创建适当的条目。如果您已经设置了 first*name cookie 为 "John"*，*last_name cookie 为 "Player"*，尝试运行 _*http://localhost:8080/ReadCookies*，将显示如下结果：

```
<tr><td>
<h2>查找 Cookies 名称和值</h2>
名称：first_name，值：John
名称：last_name，值：Player</td></tr>
```

通过 Servlet 删除 Cookies

删除 Cookies 是非常简单的。如果您想删除一个 cookie，那么您只需要按照以下三个步骤进行：

- 读取一个现有的 cookie，并把它存储在 Cookie 对象中。

- 使用 **setMaxAge()** 方法设置 cookie 的年龄为零，来删除现有的 cookie。
- 把这个 cookie 添加到响应头。

实例

下面的例子将删除现有的名为 "first_name" 的 cookie，当您下次运行 ReadCookies 的 Servlet 时，它会返回 first_name 为空值。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class DeleteCookies extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      Cookie cookie = null;
      Cookie[] cookies = null;
      // 获取与该域相关的 Cookies 的数组
      cookies = request.getCookies();

      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "Delete Cookies Example";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
                "<html>\n" +
                "<head><title>" + title + "</title></head>\n" +
                "<body bgcolor=\"#f0f0f0\">\n" );
       if( cookies != null ){
         out.println("<h2>Cookies 名称和值</h2>");
         for (int i = 0; i < cookies.length; i++){
            cookie = cookies[i];
            if((cookie.getName( )).compareTo("first_name") == 0 ){
                 cookie.setMaxAge(0);
                 response.addCookie(cookie);
                 out.print("已删除的 cookie：" +
                              cookie.getName( ) + "<br/>");
            }
            out.print("名称：" + cookie.getName( ) + ", ");
            out.print("值：" + cookie.getValue( )+" <br/>");
         }
      }else{
          out.println(
            "<h2 class="tutheader">No cookies founds</h2>");
      }
      out.println("</body>");
      out.println("</html>");
   }
}
```

编译上面的 Servlet **DeleteCookies**，并在 web.xml 文件中创建适当的条目。现在运行 *http://localhost:8080/DeleteCookies*，将显示如下结果：

```
<h2>Cookies 名称和值</h2>
已删除的 cookie：first_name
名称：first_name，值：John
名称：last_name，值：Player
```

现在尝试运行 *http://localhost:8080/ReadCookies*，它将只显示一个 cookie，如下所示：

```
<h2>查找 Cookies 名称和值</h2>
名称：last_name，值：Player
```

您可以手动在 Internet Explorer 中删除 Cookies。在"工具"菜单，选择"Internet 选项"。如果要删除所有的 Cookies，请按"删除 Cookies"。

Servlet Session 跟踪

HTTP 是一种"无状态"协议，这意味着每次客户端检索网页时，客户端打开一个单独的连接到 Web 服务器，服务器会自动不保留之前客户端请求的任何记录。

但是仍然有以下三种方式来维持 Web 客户端和 Web 服务器之间的 session 会话：

Cookies

一个 Web 服务器可以分配一个唯一的 session 会话 ID 作为每个 Web 客户端的 cookie，对于客户端的后续请求可以使用接收到的 cookie 来识别。

这可能不是一个有效的方法，因为很多浏览器不支持 cookie，所以我们建议不要使用这种方式来维持 session 会话。

隐藏的表单字段

一个 Web 服务器可以发送一个隐藏的 HTML 表单字段，以及一个唯一的 session 会话 ID，如下所示：

```
<input type="hidden" name="sessionid" value="12345">
```

该条目意味着，当表单被提交时，指定的名称和值会被自动包含在 GET 或 POST 数据中。每次当 Web 浏览器发送回请求时，session_id 值可以用于保持不同的 Web 浏览器的跟踪。

这可能是一种保持 session 会话跟踪的有效方式，但是点击常规的超文本链接（<A HREF...>）不会导致表单提交，因此隐藏的表单字段也不支持常规的 session 会话跟踪。

URL 重写

您可以在每个 URL 末尾追加一些额外的数据来标识 session 会话，服务器会把该 session 会话标识符与已存储的有关 session 会话的数据相关联。

例如，http://w3cschool.cc/file.htm;sessionid=12345，session 会话标识符被附加为 sessionid=12345，标识符可被 Web 服务器访问以识别客户端。

URL 重写是一种更好的维持 session 会话的方式，它在浏览器不支持 cookie 时能够很好地工作，但是它的缺点是会动态生成每个 URL 来为页面分配一个 session 会话 ID，即使是在很简单的静态 HTML 页面中也会如此。

HttpSession 对象

除了上述的三种方式，Servlet 还提供了 HttpSession 接口，该接口提供了一种跨多个页面请求或访问网站时识别用户以及存储有关用户信息的方式。

Servlet 容器使用这个接口来创建一个 HTTP 客户端和 HTTP 服务器之间的 session 会话。会话持续一个指定的时间段，跨多个连接或页面请求。

您会通过调用 HttpServletRequest 的公共方法 **getSession()** 来获取 HttpSession 对象，如下所示：

```
HttpSession session = request.getSession();
```

你需要在向客户端发送任何文档内容之前调用 *request.getSession()*。下面总结了 HttpSession 对象中可用的几个重要的方法：

方法	描述
public Object getAttribute(String name)	该方法返回在该 session 会话中具有指定名称的对象，如果没有指定名称的对象，则返回 null。
public Enumeration getAttributeNames()	该方法返回 String 对象的枚举，String 对象包含所有绑定到该 session 会话的对象的名称。
public long getCreationTime()	该方法返回该 session 会话被创建的时间，自格林尼治标准时间 1970 年 1 月 1 日午夜算起，以毫秒为单位。
public String getId()	该方法返回一个包含分配给该 session 会话的唯一标识符的字符串。
public long getLastAccessedTime()	该方法返回客户端最后一次发送与该 session 会话相关的请求的时间自格林尼治标准时间 1970 年 1 月 1 日午夜算起，以毫秒为单位。
public int getMaxInactiveInterval()	该方法返回 Servlet 容器在客户端访问时保持 session 会话打开的最大时间间隔，以秒为单位。
public void invalidate()	该方法指示该 session 会话无效，并解除绑定到它上面的任何对象。
public boolean isNew(	如果客户端还不知道该 session 会话，或者如果客户选择不参入该 session 会话，则该方法返回 true。
public void removeAttribute(String name)	该方法将从该 session 会话移除指定名称的对象。
public void setAttribute(String name, Object value)	该方法使用指定的名称绑定一个对象到该 session 会话。
public void setMaxInactiveInterval(int interval)	该方法在 Servlet 容器指示该 session 会话无效之前，指定客户端请求之间的时间，以秒为单位。

Session 跟踪实例

本实例说明了如何使用 HttpSession 对象获取 session 会话创建时间和最后访问时间。如果不存在 session 会话，我们将通过请求创建一个新的 session 会话。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class SessionTrack extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
```

```
        // 如果不存在 session 会话，则创建一个 session 对象
        HttpSession session = request.getSession(true);
        // 获取 session 创建时间
        Date createTime = new Date(session.getCreationTime());
        // 获取该网页的最后一次访问时间
        Date lastAccessTime =
                          new Date(session.getLastAccessedTime());

        String title = "欢迎回到我的网站";
        Integer visitCount = new Integer(0);
        String visitCountKey = new String("visitCount");
        String userIDKey = new String("userID");
        String userID = new String("ABCD");

        // 检查网页上是否有新的访问者
        if (session.isNew()){
           title = "欢迎来到我的网站";
           session.setAttribute(userIDKey, userID);
        } else {
           visitCount = (Integer)session.getAttribute(visitCountKey);
           visitCount = visitCount + 1;
           userID = (String)session.getAttribute(userIDKey);
        }
        session.setAttribute(visitCountKey,  visitCount);

        // 设置响应内容类型
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String docType =
        "<!doctype html public \"-//w3c//dtd html 4.0 " +
        "transitional//en\">\n";
        out.println(docType +
                  "<html>\n" +
                  "<head><title>" + title + "</title></head>\n" +
                  "<body bgcolor=\"#f0f0f0\">\n" +
                  "<h1 align=\"center\">" + title + "</h1>\n" +
                   "<h2 align=\"center\">Session 信息</h2>\n" +
                  "<table border=\"1\" align=\"center\">\n" +
                  "<tr bgcolor=\"#949494\">\n" +
                  "  <th>Session 信息</th><th>值</th></tr>\n" +
                  "<tr>\n" +
                  "  <td>id</td>\n" +
                  "  <td>" + session.getId() + "</td></tr>\n" +
                  "<tr>\n" +
                  "  <td>Creation Time</td>\n" +
                  "  <td>" + createTime +
                  "  </td></tr>\n" +
                  "<tr>\n" +
                  "  <td>Time of Last Access</td>\n" +
                  "  <td>" + lastAccessTime +
                  "  </td></tr>\n" +
                  "<tr>\n" +
                  "  <td>User ID</td>\n" +
                  "  <td>" + userID +
                  "  </td></tr>\n" +
                  "<tr>\n" +
                  "  <td>Number of visits</td>\n" +
                  "  <td>" + visitCount + "</td></tr>\n" +
                  "</table>\n" +
                  "</body></html>");
    }
}
```

编译上面的 Servlet **SessionTrack**，并在 web.xml 文件中创建适当的条目。在浏览器地址栏输入 *http://localhost:8080/SessionTrack*，当您第一次运行时将显示如下结果：

```
<h1>欢迎来到我的网站</h1>
<h2>Session 信息</h2>
<table>
<tbody>
<tr bgcolor="#949494"><th>Session 信息</th><th>值</th></tr>
<tr><td>id</td><td>0AE3EC93FF44E3C525B4351B77ABB2D5</td></tr>
<tr><td>Creation Time</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>Time of Last Access</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>User ID</td><td>ABCD</td></tr>
<tr><td>Number of visits</td><td>0</td></tr>
</tbody>
</table>
```

再次尝试运行相同的 Servlet，它将显示如下结果：

```
<h1>欢迎回到我的网站</h1>
<h2>Session 信息</h2>
<table>
<tbody>
<tr bgcolor="#949494"><th>Session 信息</th><th>值</th></tr>
<tr><td>id</td><td>0AE3EC93FF44E3C525B4351B77ABB2D5</td></tr>
<tr><td>Creation Time</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>Time of Last Access</td><td>Tue Jun 08 17:26:40 GMT+04:00 2014</td></tr>
<tr><td>User ID</td><td>ABCD</td></tr>
<tr><td>Number of visits</td><td>1</td></tr>
</tbody>
</table>
```

删除 Session 会话数据

当您完成了一个用户的 session 会话数据，您有以下几种选择：

- 移除一个特定的属性：您可以调用 *public void removeAttribute(String name)* 方法来删除与特定的键相关联的值。 to delete the value associated with a particular key.
- 删除整个 **session** 会话：您可以调用 *public void invalidate()* 方法来丢弃整个 session 会话。
- 设置 **session** 会话过期时间：您可以调用 *public void setMaxInactiveInterval(int interval)*

方法来单独设置 session 会话超时。
- 注销用户：如果使用的是支持 servlet 2.4 的服务器，您可以调用 **logout** 来注销 Web 服务器的客户端，并把属于所有用户的所有 session 会话设置为无效。
- **web.xml** 配置：如果您使用的是 Tomcat，除了上述方法，您还可以在 web.xml 文件中配置 session 会话超时，如下所示：

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

上面实例中的超时时间是以分钟为单位，将覆盖 Tomcat 中默认的 30 分钟超时时间。

在一个 Servlet 中的 getMaxInactiveInterval() 方法会返回 session 会话的超时时间，以秒为单位。所以，如果在 web.xml 中配置 session 会话超时时间为 15 分钟，那么 getMaxInactiveInterval() 会返回 900。

Servlet 数据库访问

本教程假定您已经了解了 JDBC 应用程序的工作方式。在您开始学习 Servlet 数据库访问之前，请确保您已经有适当的 JDBC 环境设置和数据库。

从基本概念下手，让我们来创建一个简单的表，并在表中创建几条记录。

创建数据库表

在测试数据库 **TEST** 中创建 **Employees** 表，请按以下步骤进行：

步骤 1：

打开命令行提示符（**Command Prompt**），并更改进入到安装目录，如下所示：

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

步骤 2：

登录到数据库，如下所示：

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

步骤 3：

在测试数据库 **TEST** 中创建 **Employee** 表，如下所示：

```
mysql> use TEST;
mysql> create table Employees
    (
     id int not null,
     age int not null,
     first varchar (255),
     last varchar (255)
    );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

创建数据记录

最后，在 Employee 表中创建几条记录，如下所示：

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

访问数据库

下面的实例演示了如何使用 Servlet 访问 TEST 数据库。

```
// 加载必需的库
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class DatabaseAccess extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // JDBC 驱动器名称和数据库的 URL
      static final String JDBC_DRIVER="com.mysql.jdbc.Driver";
      static final String DB_URL="jdbc:mysql://localhost/TEST";

      //  数据库的凭据
      static final String USER = "root";
      static final String PASS = "password";

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      String title = "数据库结果";
      String docType =
        "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
         out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" +
         "<h1 align=\"center\">" + title + "</h1>\n");
      try{
         // 注册 JDBC 驱动器
         Class.forName("com.mysql.jdbc.Driver");

         // 打开一个连接
         conn = DriverManager.getConnection(DB_URL,USER,PASS);

         // 执行 SQL 查询
```

```
            stmt = conn.createStatement();
            String sql;
            sql = "SELECT id, first, last, age FROM Employees";
            ResultSet rs = stmt.executeQuery(sql);

            // 从结果集中提取数据
            while(rs.next()){
               // 根据列名称检索
               int id  = rs.getInt("id");
               int age = rs.getInt("age");
               String first = rs.getString("first");
               String last = rs.getString("last");

               // 显示值
               out.println("ID: " + id + "<br>");
               out.println(", Age: " + age + "<br>");
               out.println(", First: " + first + "<br>");
               out.println(", Last: " + last + "<br>");
            }
            out.println("</body></html>");

            // 清理环境
            rs.close();
            stmt.close();
            conn.close();
         }catch(SQLException se){
            // 处理 JDBC 错误
            se.printStackTrace();
         }catch(Exception e){
            // 处理 Class.forName 错误
            e.printStackTrace();
         }finally{
            // 最后是用于关闭资源的块
            try{
               if(stmt!=null)
                  stmt.close();
            }catch(SQLException se2){
            }// 我们不能做什么
            try{
               if(conn!=null)
               conn.close();
            }catch(SQLException se){
               se.printStackTrace();
            }//end finally try
         } //end try
      }
   }
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>DatabaseAccess</servlet-name>
     <servlet-class>DatabaseAccess</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>DatabaseAccess</servlet-name>
     <url-pattern>/DatabaseAccess</url-pattern>
 </servlet-mapping>
....
```

现在调用这个 Servlet，输入链接：http://localhost:8080/DatabaseAccess，将显示以下响应结果：

```
<h1 align="center">数据库结果</h1>
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
```

Servlet 文件上传

Servlet 可以与 HTML form 标签一起使用，来允许用户上传文件到服务器。上传的文件可以是文本文件或图像文件或任何文档。

创建一个文件上传表单

下面的 HTML 代码创建了一个文件上传表单。以下几点需要注意：

- 表单 **method** 属性应该设置为 **POST** 方法，不能使用 GET 方法。
- 表单 **enctype** 属性应该设置为 **multipart/form-data**.
- 表单 **action** 属性应该设置为在后端服务器上处理文件上传的 Servlet 文件。下面的实例使用了 **UploadServlet** Servlet 来上传文件。
- 上传单个文件，您应该使用单个带有属性 type="file" 的 <input .../> 标签。为了允许多个文件上传，请包含多个 name 属性值不同的 input 标签。输入标签具有不同的名称属性的值。浏览器会为每个 input 标签关联一个浏览按钮。

```
<html>
<head>
<title>文件上传表单</title>
</head>
<body>
<h3>文件上传：</h3>
请选择要上传的文件：<br />
<form action="UploadServlet" method="post"
                        enctype="multipart/form-data">
<input type="file" name="file" size="50" />
<br />
<input type="submit" value="上传文件" />
</form>
</body>
</html>
```

这将显示下面的结果，允许用户从本地计算机选择一个文件，当用户点击"上传文件"时，表单会连同从本地计算机选择的文件一起提交：

```
<b>文件上传：</b>
请选择要上传的文件：<br />
<input type="file" name="file" size="50" />
<br />
<input type="button" value="上传文件" />
<br />
注：这只是虚拟的表单，不会正常工作。
```

编写后台 Servlet

以下是 Servlet **UploadServlet**，会接受上传的文件，并把它储存在目录 <Tomcat-installation-directory>/webapps/data 中。这个目录名也可以使用外部配置来添加，比如 web.xml 中的 **context-param** 元素，如下所示：

```
<web-app>
....
<context-param>
    <description>Location to store uploaded file</description>
    <param-name>file-upload</param-name>
    <param-value>
         c:\apache-tomcat-5.5.29\webapps\data\
     </param-value>
</context-param>
....
</web-app>
```

以下是 UploadServlet 的源代码，可以一次处理多个文件的上传。在继续操作之前，请确认下列各项：

- 下面的实例依赖于 FileUpload，所以一定要确保在您的 classpath 中有最新版本的 **commons-fileupload.x.x.jar** 文件。可以从 http://commons.apache.org/fileupload/ 下载。
- FileUpload 依赖于 Commons IO，所以一定要确保在您的 classpath 中有最新版本的 **commons-io-x.x.jar** 文件。可以从 http://commons.apache.org/io/ 下载。
- 在测试下面实例时，您上传的文件大小不能大于 *maxFileSize*，否则文件将无法上传。
- 请确保已经提前创建好目录 c:\temp and c:\apache-tomcat-5.5.29\webapps\data。

```
// 导入必需的 java 库
import java.io.*;
import java.util.*;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.FileUploadException;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;
import org.apache.commons.io.output.*;

public class UploadServlet extends HttpServlet {

   private boolean isMultipart;
   private String filePath;
   private int maxFileSize = 50 * 1024;
   private int maxMemSize = 4 * 1024;
   private File file ;

   public void init( ){
      // 获取文件将被存储的位置
      filePath =
             getServletContext().getInitParameter("file-upload");
   }
   public void doPost(HttpServletRequest request,
               HttpServletResponse response)
              throws ServletException, java.io.IOException {
```

```
      // 检查我们有一个文件上传请求
      isMultipart = ServletFileUpload.isMultipartContent(request);
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter( );
      if( !isMultipart ){
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet upload</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<p>No file uploaded</p>");
         out.println("</body>");
         out.println("</html>");
         return;
      }
      DiskFileItemFactory factory = new DiskFileItemFactory();
      // 文件大小的最大值将被存储在内存中
      factory.setSizeThreshold(maxMemSize);
      // Location to save data that is larger than maxMemSize.
      factory.setRepository(new File("c:\\temp"));

      // 创建一个新的文件上传处理程序
      ServletFileUpload upload = new ServletFileUpload(factory);
      // 允许上传的文件大小的最大值
      upload.setSizeMax( maxFileSize );

      try{
      // 解析请求，获取文件项
      List fileItems = upload.parseRequest(request);

      // 处理上传的文件项
      Iterator i = fileItems.iterator();

      out.println("<html>");
      out.println("<head>");
      out.println("<title>Servlet upload</title>");
      out.println("</head>");
      out.println("<body>");
      while ( i.hasNext () )
      {
         FileItem fi = (FileItem)i.next();
         if ( !fi.isFormField () )
         {
            // 获取上传文件的参数
            String fieldName = fi.getFieldName();
            String fileName = fi.getName();
            String contentType = fi.getContentType();
            boolean isInMemory = fi.isInMemory();
            long sizeInBytes = fi.getSize();
            // 写入文件
            if( fileName.lastIndexOf("\\") >= 0 ){
               file = new File( filePath +
               fileName.substring( fileName.lastIndexOf("\\"))) ;
            }else{
               file = new File( filePath +
               fileName.substring(fileName.lastIndexOf("\\")+1)) ;
            }
            fi.write( file ) ;
            out.println("Uploaded Filename: " + fileName + "<br>");
         }
      }
      out.println("</body>");
      out.println("</html>");
   }catch(Exception ex) {
       System.out.println(ex);
   }
   }
   public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, java.io.IOException {

        throw new ServletException("GET method used with " +
```

```
                getClass( ).getName( )+": POST method required.");
   }
}
```

编译和运行 Servlet

编译上面的 Servlet UploadServlet，并在 web.xml 文件中创建所需的条目，如下所示：

```
<servlet>
   <servlet-name>UploadServlet</servlet-name>
   <servlet-class>UploadServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>UploadServlet</servlet-name>
   <url-pattern>/UploadServlet</url-pattern>
</servlet-mapping>
```

现在尝试使用您在上面创建的 HTML 表单来上传文件。当您在浏览器中访问：http://localhost:8080/UploadFile.htm 时，它会显示下面的结果，这将有助于您从本地计算机上传任何文件。

```
<b>文件上传：</b>
请选择要上传的文件：<br />
<input type="file" name="file" size="50" />
<br />
<input type="button" value="上传文件" />
```

如果您的 Servelt 脚本能正常工作，那么您的文件会被上传到 c:\apache-tomcat-5.5.29\webapps\data\ 目录中。

Servlet 处理日期

使用 Servlet 的最重要的优势之一是，可以使用核心 Java 中的大多数可用的方法。本章将讲解 Java 提供的 **java.util** 包中的 **Date** 类，这个类封装了当前的日期和时间。

Date 类支持两个构造函数。第一个构造函数初始化当前日期和时间的对象。

```
Date( )
```

下面的构造函数接受一个参数，该参数等于 1970 年 1 月 1 日午夜以来经过的毫秒数。

```
Date(long millisec)
```

一旦您有一个可用的 Date 对象，您可以调用下列任意支持的方法来使用日期：

方法	描述
boolean after(Date date)	如果调用的 Date 对象中包含的日期在 date 指定的日期之后，则返回 true，否则返回 false。
boolean before(Date date)	如果调用的 Date 对象中包含的日期在 date 指定的日期之前，则返回 true，否则返回 false。
Object clone( )	重复调用 Date 对象。
int compareTo(Date date)	把调用对象的值与 date 的值进行比较。如果两个值是相等的，则返回 0。如果调用对象在 date 之前，则返回一个负值。如果调用对象在 date 之后，则返回一个正值。
int compareTo(Object obj)	如果 obj 是 Date 类，则操作等同于 compareTo(Date)。否则，它会抛出一个 ClassCastException。
boolean equals(Object date)	如果调用的 Date 对象中包含的时间和日期与 date 指定的相同，则返回 true，否则返回 false。
long getTime( )	返回 1970 年 1 月 1 日以来经过的毫秒数。
int hashCode( )	为调用对象返回哈希代码。
void setTime(long time)	设置 time 指定的时间和日期，这表示从 1970 年 1 月 1 日午夜以来经过的时间（以毫秒为单位）。
String toString( )	转换调用的 Date 对象为一个字符串，并返回结果。

获取当前的日期和时间

在 Java Servlet 中获取当前的日期和时间是非常容易的。您可以使用一个简单的 Date 对象的 *toString()* 方法来输出当前的日期和时间，如下所示：

```
// 导入必需的 java 库
import java.io.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class CurrentDate extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "显示当前的日期和时间";
      Date date = new Date();
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">" + date.toString() + "</h2>\n" +
        "</body></html>");
  }
}
```

现在，让我们来编译上面的 Servlet，并在 web.xml 文件中创建适当的条目，然后通过访问 http://localhost:8080/CurrentDate 来调用该 Servlet。这将会产生如下的结果：

```
<h1>显示当前的日期和时间</h1>

<h2>Mon Jun 21 21:46:49 GMT+04:00 2010</h2>
```

尝试刷新 URL http://localhost:8080/CurrentDate，每隔几秒刷新一次您都会发现显示时间的差异。

日期比较

正如上面所提到的，您可以在 Servlet 中使用所有可用的 Java 方法。如果您需要比较两个日期，以下是方法：

- 您可以使用 getTime() 来获取两个对象自 1970 年 1 月 1 日午夜以来经过的时间（以毫秒为单位），然后对这两个值进行比较。
- 您可以使用方法 before( )、after( ) 和 equals( )。由于一个月里 12 号在 18 号之前，例如，new Date(99, 2, 12).before(new Date (99, 2, 18)) 返回 true。

- 您可以使用 compareTo( ) 方法，该方法由 Comparable 接口定义，由 Date 实现。

使用 SimpleDateFormat 格式化日期

SimpleDateFormat 是一个以语言环境敏感的方式来格式化和解析日期的具体类。SimpleDateFormat 允许您选择任何用户定义的日期时间格式化的模式。

让我们修改上面的实例，如下所示：

```
// 导入必需的 java 库
import java.io.*;
import java.text.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

// 扩展 HttpServlet 类
public class CurrentDate extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      PrintWriter out = response.getWriter();
      String title = "显示当前的日期和时间";
      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">" + ft.format(dNow) + "</h2>\n" +
        "</body></html>");
  }
}
```

再次编译上面的 Servlet，然后通过访问 http://localhost:8080/CurrentDate 来调用该 Servlet。这将会产生如下的结果：

```
<h1>显示当前的日期和时间</h1>

<h2>Mon 2010.06.21 at 10:06:44 PM GMT+04:00</h2>
```

简单的日期格式的格式代码

使用事件模式字符串来指定时间格式。在这种模式下，所有的 ASCII 字母被保留为模式字母，这些字母定义如下：

字符	描述	实例
G	Era 指示器	AD
y	四位数表示的年	2001
M	一年中的月	July 或 07
d	一月中的第几天	10
h	带有 A.M./P.M. 的小时（1~12）	12
H	一天中的第几小时（0~23）	22
m	一小时中的第几分	30
s	一分中的第几秒	55
S	毫秒	234
E	一周中的星期几	Tuesday
D	一年中的第几天	360
F	所在的周是这个月的第几周	2 (second Wed. in July)
w	一年中的第几周	40
W	一月中的第几周	1
a	A.M./P.M. 标记	PM
k	一天中的第几小时（1~24）	24
K	带有 A.M./P.M. 的小时（0~11）	10
z	时区	Eastern Standard Time
'	Escape for text	Delimiter
"	单引号	`

如需查看可用的处理日期方法的完整列表，您可以参考标准的 Java 文档。

Servlet 网页重定向

当文档移动到新的位置，我们需要向客户端发送这个新位置时，我们需要用到网页重定向。当然，也可能是为了负载均衡，或者只是为了简单的随机，这些情况都有可能用到网页重定向。

重定向请求到另一个网页的最简单的方式是使用 response 对象的 **sendRedirect()** 方法。下面是该方法的定义：将请求重定向到另一页的最简单的方法是，用方法的sendRedirect()的响应对象。以下是这种方法的定义：

```
public void HttpServletResponse.sendRedirect(String location)
throws IOException
```

该方法把响应连同状态码和新的网页位置发送回浏览器。您也可以通过把 setStatus() 和 setHeader() 方法一起使用来达到同样的效果：

```
....
String site = "http://www.newpage.com" ;
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
....
```

实例

本实例显示了 Servlet 如何进行页面重定向到另一个位置：

```
import java.io.*;
import java.sql.Date;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PageRedirect extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      // 要重定向的新位置
      String site = new String("http://www.w3cschool.cc");

      response.setStatus(response.SC_MOVED_TEMPORARILY);
      response.setHeader("Location", site);
    }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>PageRedirect</servlet-name>
     <servlet-class>PageRedirect</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>PageRedirect</servlet-name>
     <url-pattern>/PageRedirect</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/PageRedirect 来调用这个 Servlet。这将把您转到给定的 URL http://www.w3cschool.cc 。

Servlet 点击计数器

网页点击计数器

很多时候，您可能有兴趣知道网站的某个特定页面上的总点击量。使用 Servlet 来计算这些点击量是非常简单的，因为一个 Servlet 的生命周期是由它运行所在的容器控制的。

以下是实现一个简单的基于 Servlet 生命周期的网页点击计数器需要采取的步骤：

- 在 init() 方法中初始化一个全局变量。
- 每次调用 doGet() 或 doPost() 方法时，都增加全局变量。
- 如果需要，您可以使用一个数据库表来存储全局变量的值在 destroy() 中。在下次初始化 Servlet 时，该值可在 init() 方法内被读取。这一步是可选的。
- 如果您只想对一个 session 会话计数一次页面点击，那么请使用 isNew() 方法来检查该 session 会话是否已点击过相同页面。这一步是可选的。
- 您可以通过显示全局计数器的值，来在网站上展示页面的总点击量。这一步是可选的。

在这里，我们假设 Web 容器将无法重新启动。如果是重新启动或 Servlet 被销毁，计数器将被重置。

实例

本实例演示了如何实现一个简单的网页点击计数器：

```
import java.io.*;
import java.sql.Date;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PageHitCounter extends HttpServlet{

  private int hitCount;

  public void init()
  {
     // 重置点击计数器
     hitCount = 0;
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");
      // 该方法在 Servlet 被点击时执行
      // 增加 hitCount
      hitCount++;
      PrintWriter out = response.getWriter();
      String title = "总点击量";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">" + hitCount + "</h2>\n" +
        "</body></html>");

  }
  public void destroy()
  {
      // 这一步是可选的，但是如果需要，您可以把 hitCount 的值写入到数据库
  }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>PageHitCounter</servlet-name>
     <servlet-class>PageHitCounter</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>PageHitCounter</servlet-name>
     <url-pattern>/PageHitCounter</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/PageHitCounter 来调用这个 Servlet。这将会在每次页面刷新时，把计数器的值增加 1，结果如下所示：

```
<h1>总点击量</h1>

<h2>6</h2>
```

网站点击计数器

很多时候，您可能有兴趣知道整个网站的总点击量。在 Servlet 中，这也是非常简单的，我们可以使用过滤器做到这一点。

以下是实现一个简单的基于过滤器生命周期的网站点击计数器需要采取的步骤：

- 在过滤器的 init() 方法中初始化一个全局变量。
- 每次调用 doFilter 方法时，都增加全局变量。
- 如果需要，您可以使用一个数据库表来存储全局变量的值在过滤器的 destroy() 中。在下次初始化过滤器时，该值可在 init() 方法内被读取。这一步是可选的。

在这里，我们假设 Web 容器将无法重新启动。如果是重新启动或 Servlet 被销毁，点击计数器将被重置。

实例

本实例演示了如何实现一个简单的网站点击计数器：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class SiteHitCounter implements Filter{

  private int hitCount;

  public void  init(FilterConfig config)
                    throws ServletException{
     // 重置点击计数器
     hitCount = 0;
  }

  public void  doFilter(ServletRequest request,
              ServletResponse response,
              FilterChain chain)
              throws java.io.IOException, ServletException {

      // 把计数器的值增加 1
      hitCount++;

      // 输出计数器
      System.out.println("网站访问统计："+ hitCount );

      // 把请求传回到过滤器链
      chain.doFilter(request,response);
  }
  public void destroy()
  {
      // 这一步是可选的，但是如果需要，您可以把 hitCount 的值写入到数据库
  }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
<filter>
   <filter-name>SiteHitCounter</filter-name>
   <filter-class>SiteHitCounter</filter-class>
</filter>

<filter-mapping>
   <filter-name>SiteHitCounter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

....
```

现在访问网站的任意页面，比如 http://localhost:8080/ 。这将会在每次任意页面被点击时，把计数器的值增加 1，它会在日志中显示以下消息：

```
网站访问统计： 1
网站访问统计： 2
网站访问统计： 3
网站访问统计： 4
网站访问统计： 5
..................
```

Servlet 自动刷新页面

假设有一个网页，它是显示现场比赛成绩或股票市场状况或货币兑换率。对于所有这些类型的页面，您需要定期刷新网页。

Java Servlet 提供了一个机制，使得网页会在给定的时间间隔自动刷新。

刷新网页的最简单的方式是使用响应对象的方法 **setIntHeader()**。以下是这种方法的定义：

```
public void setIntHeader(String header, int headerValue)
```

此方法把头信息 "Refresh" 连同一个表示时间间隔的整数值（以秒为单位）发送回浏览器。

自动刷新页面实例

本实例演示了 Servlet 如何使用 **setIntHeader()** 方法来设置 **Refresh** 头信息，从而实现自动刷新页面。

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 扩展 HttpServlet 类
public class Refresh extends HttpServlet {

  // 处理 GET 方法请求的方法
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置刷新自动加载的事件间隔为 5 秒
      response.setIntHeader("Refresh", 5);

      // 设置响应内容类型
      response.setContentType("text/html");

      // 获取当前的时间
      Calendar calendar = new GregorianCalendar();
      String am_pm;
      int hour = calendar.get(Calendar.HOUR);
      int minute = calendar.get(Calendar.MINUTE);
      int second = calendar.get(Calendar.SECOND);
      if(calendar.get(Calendar.AM_PM) == 0)
        am_pm = "AM";
      else
        am_pm = "PM";

      String CT = hour+":"+ minute +":"+ second +" "+ am_pm;

      PrintWriter out = response.getWriter();
      String title = "使用 Servlet 自动刷新页面";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n"+
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<p>当前时间是：" + CT + "</p>\n");
  }
  // 处理 POST 方法请求的方法
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
     doGet(request, response);
  }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>Refresh</servlet-name>
     <servlet-class>Refresh</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>Refresh</servlet-name>
     <url-pattern>/Refresh</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/Refresh 来调用这个 Servlet。这将会每隔 5 秒钟显示一次当前系统时间。运行该 Servlet，并等待查看结果：

```
<h1>使用 Servlet 自动刷新页面</h1>

当前时间是：9:44:50 PM
```

Servlet 发送电子邮件

使用 Servlet 发送一封电子邮件是很简单的，但首先您必须在您的计算机上安装 **JavaMail API** 和 **Java Activation Framework**）**JAF**）。

- 您可以从 Java 标准网站下载最新版本的 JavaMail（版本 1.2）。
- 您可以从 Java 标准网站下载最新版本的 JAF（版本 1.1.1）。

下载并解压缩这些文件，在新创建的顶层目录中，您会发现这两个应用程序的一些 jar 文件。您需要把 **mail.jar** 和 **activation.jar** 文件添加到您的 CLASSPATH 中。

发送一封简单的电子邮件

下面的实例将从您的计算机上发送一封简单的电子邮件。这里假设您的本地主机已连接到互联网，并支持发送电子邮件。同时确保 Java Email API 包和 JAF 包的所有的 jar 文件在 CLASSPATH 中都是可用的。

```
// 文件名 SendEmail.java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 收件人的电子邮件 ID
      String to = "abcd@gmail.com";

      // 发件人的电子邮件 ID
      String from = "web@gmail.com";

      // 假设您是从本地主机发送电子邮件
      String host = "localhost";

      // 获取系统的属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象
      Session session = Session.getDefaultInstance(properties);

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      try{
         // 创建一个默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);
         // 设置 From: header field of the header.
         message.setFrom(new InternetAddress(from));
         // 设置 To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));
         // 设置 Subject: header field
         message.setSubject("This is the Subject Line!");
         // 现在设置实际消息
         message.setText("This is actual message");
         // 发送消息
         Transport.send(message);
         String title = "发送电子邮件";
         String res = "成功发送消息...";
         String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
         out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" +
         "<h1 align=\"center\">" + title + "</h1>\n" +
         "<p align=\"center\">" + res + "</p>\n" +
         "</body></html>");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

现在让我们来编译上面的 Servlet，并在 web.xml 文件中创建以下条目：

```
....
 <servlet>
     <servlet-name>SendEmail</servlet-name>
     <servlet-class>SendEmail</servlet-class>
 </servlet>

 <servlet-mapping>
     <servlet-name>SendEmail</servlet-name>
     <url-pattern>/SendEmail</url-pattern>
 </servlet-mapping>
....
```

现在通过访问 URL http://localhost:8080/SendEmail 来调用这个 Servlet。这将会发送一封电子邮件到给定的电子邮件 ID *abcd@gmail.com*，并将显示下面所示的响应：

```
<h1>发送电子邮件</h1>

成功发送消息...
```

如果您想要发送一封电子邮件给多个收件人，那么请使用下面的方法来指定多个电子邮件 ID：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

下面是对参数的描述：

- **type**：这将被设置为 TO、CC 或 BCC。在这里，CC 代表抄送，BCC 代表密件抄送。例如 *Message.RecipientType.TO*。
- **addresses**：这是电子邮件 ID 的数组。当指定电子邮件 ID 时，您需要使用 InternetAddress() 方法。

发送一封 HTML 电子邮件

下面的实例将从您的计算机上发送一封 HTML 格式的电子邮件。这里假设您的本地主机已连接到互联网，并支持发送电子邮件。同时确保 Java Email API 包和 JAF 包的所有的 jar 文件在 CLASSPATH 中都是可用的。

本实例与上一个实例很类似，但是这里我们使用 setContent() 方法来设置第二个参数为 "text/html" 的内容，该参数用来指定 HTML 内容是包含在消息中的。

使用这个实例，您可以发送内容大小不限的 HTML 内容。

```
// 文件名 SendEmail.java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 收件人的电子邮件 ID
      String to = "abcd@gmail.com";

      // 发件人的电子邮件 ID
      String from = "web@gmail.com";

      // 假设您是从本地主机发送电子邮件
      String host = "localhost";

      // 获取系统的属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象
      Session session = Session.getDefaultInstance(properties);

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      try{
         // 创建一个默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);
         // 设置 From: header field of the header.
         message.setFrom(new InternetAddress(from));
         // 设置 To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));
         // 设置 Subject: header field
         message.setSubject("This is the Subject Line!");

         // 设置实际的 HTML 消息，内容大小不限
         message.setContent("<h1>This is actual message</h1>",
                            "text/html" );
         // 发送消息
         Transport.send(message);
         String title = "发送电子邮件";
         String res = "成功发送消息...";
         String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
         out.println(docType +
         "<html>\n" +
         "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" +
         "<h1 align=\"center\">" + title + "</h1>\n" +
         "<p align=\"center\">" + res + "</p>\n" +
         "</body></html>");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

编译并运行上面的 Servlet，在给定的电子邮件 ID 上发送 HTML 消息。

在电子邮件中发送附件

下面的实例将从您的计算机上发送一封带有附件的电子邮件。这里假设您的本地主机已连接到互联网，并支持发送电子邮件。同时确保 Java Email API 包和 JAF 包的所有的 jar 文件在 CLASSPATH 中都是可用的。

```
// 文件名 SendEmail.java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 收件人的电子邮件 ID
      String to = "abcd@gmail.com";

      // 发件人的电子邮件 ID
      String from = "web@gmail.com";

      // 假设您是从本地主机发送电子邮件
      String host = "localhost";

      // 获取系统的属性
      Properties properties = System.getProperties();

      // 设置邮件服务器
      properties.setProperty("mail.smtp.host", host);

      // 获取默认的 Session 对象
      Session session = Session.getDefaultInstance(properties);

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

       try{
         // 创建一个默认的 MimeMessage 对象
         MimeMessage message = new MimeMessage(session);

         // 设置 From: header field of the header.
         message.setFrom(new InternetAddress(from));

         // 设置 To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                              new InternetAddress(to));

         // 设置 Subject: header field
         message.setSubject("This is the Subject Line!");

         // 创建消息部分
         BodyPart messageBodyPart = new MimeBodyPart();

         // 填写消息
         messageBodyPart.setText("This is message body");
```

```
            // 创建一个多部分消息
            Multipart multipart = new MimeMultipart();

            // 设置文本消息部分
            multipart.addBodyPart(messageBodyPart);

            // 第二部分是附件
            messageBodyPart = new MimeBodyPart();
            String filename = "file.txt";
            DataSource source = new FileDataSource(filename);
            messageBodyPart.setDataHandler(new DataHandler(source));
            messageBodyPart.setFileName(filename);
            multipart.addBodyPart(messageBodyPart);

            // 发送完整的消息部分
            message.setContent(multipart );

            // 发送消息
            Transport.send(message);
            String title = "发送电子邮件";
            String res = "成功发送电子邮件...";
            String docType =
            "<!doctype html public \"-//w3c//dtd html 4.0 " +
            "transitional//en\">\n";
            out.println(docType +
            "<html>\n" +
            "<head><title>" + title + "</title></head>\n" +
            "<body bgcolor=\"#f0f0f0\">\n" +
            "<h1 align=\"center\">" + title + "</h1>\n" +
            "<p align=\"center\">" + res + "</p>\n" +
            "</body></html>");
         }catch (MessagingException mex) {
            mex.printStackTrace();
         }
      }
   }
```

编译并运行上面的 Servlet，在给定的电子邮件 ID 上发送带有文件附件的消息。

用户身份认证部分

如果需要向电子邮件服务器提供用户 ID 和密码进行身份认证，那么您可以设置如下属性：

```
props.setProperty("mail.user", "myuser");
props.setProperty("mail.password", "mypwd");
```

电子邮件发送机制的其余部分与上面讲解的保持一致。

Servlet 包

涉及到 WEB-INF 子目录的 Web 应用程序结构是所有的 Java web 应用程序的标准，并由 Servlet API 规范指定。给定一个顶级目录名 myapp，目录结构如下所示：

```
/myapp
    /images
    /WEB-INF
        /classes
        /lib
```

WEB-INF 子目录中包含应用程序的部署描述符，名为 web.xml。所有的 HTML 文件都位于顶级目录 *myapp* 下。对于 admin 用户，您会发现 ROOT 目录是 myApp 的父目录。

创建包中的 Servlet

WEB-INF/classes 目录包含了所有的 Servlet 类和其他类文件，类文件所在的目录结构与他们的包名称匹配。例如，如果您有一个完全合格的类名称 **com.myorg.MyServlet**，那么这个 Servlet 类必须位于以下目录中：

```
/myapp/WEB-INF/classes/com/myorg/MyServlet.class
```

下面的例子创建包名为 *com.myorg* 的 MyServlet 类。

```
// 为包命名
package com.myorg;

// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

  private String message;

  public void init() throws ServletException
  {
      // 执行必需的的初始化
      message = "Hello World";
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 设置响应内容类型
      response.setContentType("text/html");

      // 实际的逻辑是在这里
      PrintWriter out = response.getWriter();
      out.println("<h1>" + message + "</h1>");
  }

  public void destroy()
  {
      // 什么也不做
  }
}
```

编译包中的 Servlet

编译包中的类与编译其他的类没有什么大的不同。最简单的方法是让您的 java 文件保留完全限定路径，如上面提到的类，将被保留在 com.myorg 中。您还需要在 CLASSPATH 中添加该目录。

假设您的环境已正确设置，进入 **<Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes** 目录，并编译 MyServlet.java，如下所示：

```
$ javac MyServlet.java
```

如果 Servlet 依赖于其他库，那么您必须在 CLASSPATH 中也要引用那些 JAR 文件。这里我只引用了 servlet-api.jar JAR 文件，因为我在 Hello World 程序中并没有使用任何其他库。

该命令行使用内置的 javac 编译器，它是 Sun Microsystems Java 软件开发工具包（JDK，全称 Java Software Development Kit）附带的。 Microsystems的Java软件开发工具包（JDK）。为了让该命令正常工作，必须包括您在 PATH 环境变量中所使用的 Java SDK 的位置。

如果一切顺利，上述编译会在同一目录下生成 **MyServlet.class** 文件。下一节将解释如何部把一个已编译的 Servlet 部署到生产中。

Servlet 打包部署

默认情况下，Servlet 应用程序位于路径 <Tomcat-installation-directory>/webapps/ROOT 下，且类文件放在 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes 中。

如果您有一个完全合格的类名称 **com.myorg.MyServlet**，那么这个 Servlet 类必须位于 WEB-INF/classes/com/myorg/MyServlet.class 中，您需要在位于 <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/ 的 web.xml 文件中创建以下条目：

```
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>com.myorg.MyServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>/MyServlet</url-pattern>
</servlet-mapping>
```

上面的条目要被创建在 web.xml 文件中的 <web-app>...</web-app> 标签内。在该文件中可能已经有各种可用的条目，但不要在意。

到这里，您基本上已经完成了，现在让我们使用 <Tomcat-installation-directory>\bin\startup.bat（在 Windows 上）或 <Tomcat-installation-directory>/bin/startup.sh（在 Linux/Solaris 等上）启动 tomcat 服务器，最后在浏览器的地址栏中输入 **http://localhost:8080/MyServlet**。如果一切顺利，您会看到下面的结果：

```
<h1>Hello World</h1>
```

Servlet 调试

测试/调试 Servlet 始终是开发使用过程中的难点。Servlet 往往涉及大量的客户端/服务器交互，可能会出现错误但又难以重现。

这里有一些提示和建议，可以帮助您调试。

System.out.println()

System.out.println() 是作为一个标记来使用的，用来测试一段特定的代码是否被执行。我们也可以打印出变量的值。此外：

- 由于 System 对象是核心 Java 对象的一部分，它可以在不需要安装任何额外类的情况下被用于任何地方。这包括 Servlet、JSP、RMI、EJB's、普通的 Beans 和类，以及独立的应用程序。
- 与在断点处停止不同，写入到 System.out 不会干扰到应用程序的正常执行流程，这使得它在时序是至关重要的时候显得尤为有价值。

下面是使用 System.out.println() 的语法：

```
System.out.println("Debugging message");
```

通过上面的语法生成的所有消息将被记录在 Web 服务器日志文件中。

消息日志

使用适当的日志记录方法来记录所有调试、警告和错误消息，这是非常好的想法，推荐使用 log4J 来记录所有的消息。

Servlet API 还提供了一个简单的输出信息的方式，使用 log() 方法，如下所示：

```
// 导入必需的 java 库
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ContextLog extends HttpServlet {
  public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
         java.io.IOException {

      String par = request.getParameter("par1");
      // 调用两个 ServletContext.log 方法
      ServletContext context = getServletContext( );

      if (par == null || par.equals(""))
      // 通过 Throwable 参数记录版本
      context.log("No message received:",
          new IllegalStateException("Missing parameter"));
      else
          context.log("Here is the visitor's message: " + par);

      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter( );
      String title = "Context Log";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n" +
        "<h2 align=\"center\">Messages sent</h2>\n" +
        "</body></html>");
    } //doGet
}
```

ServletContext 把它的文本消息记录到 Servlet 容器的日志文件中。对于 Tomcat，这些日志可以在 <Tomcat-installation-directory>/logs 目录中找到。

这些日志文件确实对新出现的错误或问题的频率给出指示。正因为如此，建议在通常不会发生的异常的 catch 子句中使用 log() 函数。

使用 JDB 调试器

您可以使用调试 applet 或应用程序的 jdb 命令来调试 Servlet。

为了调试一个 Servlet，我们可以调试 sun.servlet.http.HttpServer，然后把它看成是 HttpServer 执行 Servlet 来响应浏览器端的 HTTP 请求。这与调试 applet 小程序非常相似。与调试 applet 不同的是，实际被调试的程序是 sun.applet.AppletViewer。

大多数调试器会自动隐藏如何调试 applet 的细节。同样的，对于 servlet，您必须帮调试器执行以下操作：

- 设置您的调试器的类路径 classpath，以便它可以找到 sun.servlet.http.Http-Server 和相关的类。
- 设置您的调试器的类路径 classpath，以便它可以找到您的 servlet 和支持的类，通常是

在 server_root/servlets 和 server_root/classes 中。

您通常不会希望 server_root/servlets 在您的 classpath 中，因为它会禁用 servlet 的重新加载。但是这种包含规则对于调试是非常有用的。它允许您的调试器在 HttpServer 中的自定义 Servlet 加载器加载 Servlet 之前在 Servlet 中设置断点。

如果您已经设置了正确的类路径 classpath，就可以开始调试 sun.servlet.http.HttpServer。可以在您想要调试的 Servlet 代码中设置断点，然后通过 Web 浏览器使用给定的 Servlet（http://localhost:8080/servlet/ServletToDebug）向 HttpServer 发出请求。您会看到程序执行到断点处会停止。

使用注释

代码中的注释有助于以各种方式进行调试。注释可用于调试过程的很多其他方式中。

该 Servlet 使用 Java 注释和单行注释（//...），多行注释（/ .../）可用于暂时移除部分 Java 代码。如果 bug 消失，仔细看看您刚才注释的代码并找出问题所在。

客户端和服务器端头信息

有时，当一个 Servlet 并没有像预期那样时，查看原始的 HTTP 请求和响应是非常有用的。如果您熟悉 HTTP 结构，您可以阅读请求和响应，看看这些头信息究竟是什么。

重要的调试技巧

下面列出了一些 Servlet 调试的技巧：

- 请注意，server_root/classes 不会重载，而 server_root/servlets 可能会。
- 要求浏览器显示它所显示的页面的原始内容。这有助于识别格式的问题。它通常是"视图"菜单下的一个选项。
- 通过强制执行完全重新加载页面来确保浏览器还没有缓存前一个请求的输出。在 Netscape Navigator 中，请使用 Shift-Reload，在 Internet Explorer 中，请使用 Shift-Refresh。
- 请确认 servlet 的 init() 方法接受一个 ServletConfig 参数，并调用 super.init(config)。

Servlet 国际化

在我们开始之前，先来看看三个重要术语：

- 国际化（**i18n**）：这意味着一个网站提供了不同版本的翻译成访问者的语言或国籍的内容。
- 本地化（**l10n**）：这意味着向网站添加资源，以使其适应特定的地理或文化区域，例如网站翻译成印地文（Hindi）。
- 区域设置（**locale**）：这是一个特殊的文化或地理区域。它通常指语言符号后跟一个下划线和一个国家符号。例如 "en_US" 表示针对 US 的英语区域设置。

当建立一个全球性的网站时有一些注意事项。本教程不会讲解这些注意事项的完整细节，但它会通过一个很好的实例向您演示如何通过差异化定位（即区域设置）来让网页以不同语言呈现。

Servlet 可以根据请求者的区域设置拾取相应版本的网站，并根据当地的语言、文化和需求提供相应的网站版本。以下是 request 对象中返回 Locale 对象的方法。

```
java.util.Locale request.getLocale()
```

检测区域设置

下面列出了重要的区域设置方法，您可以使用它们来检测请求者的地理位置、语言和区域设置。下面所有的方法都显示了请求者浏览器中设置的国家名称和语言名称。

方法	描述
String getCountry()	该方法以 2 个大写字母形式的 ISO 3166 格式返回该区域设置的国家/地区代码。
String getDisplayCountry()	该方法返回适合向用户显示的区域设置的国家的名称。
String getLanguage()	该方法以小写字母形式的 ISO 639 格式返回该区域设置的语言代码。
String getDisplayLanguage()	该方法返回适合向用户显示的区域设置的语言的名称。
String getISO3Country()	该方法返回该区域设置的国家的三个字母缩写。
String getISO3Language()	该方法返回该区域设置的语言的三个字母的缩写。

实例

本实例演示了如何显示某个请求的语言和相关的国家：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;

public class GetLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
      // 获取客户端的区域设置
      Locale locale = request.getLocale();
      String language = locale.getLanguage();
      String country = locale.getCountry();

      // 设置响应内容类型
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      String title = "检测区域设置";
      String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + language + "</h1>\n" +
        "<h2 align=\"center\">" + country + "</h2>\n" +
        "</body></html>");
  }
}
```

语言设置

Servlet 可以输出以西欧语言（如英语、西班牙语、德语、法语、意大利语、荷兰语等）编写的页面。在这里，为了能正确显示所有的字符，设置 Content-Language 头是非常重要的。

第二点是使用 HTML 实体显示所有的特殊字符，例如，"ñ" 表示 "?"，"¡" 表示 "?"，如下所示：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;

public class DisplaySpanish extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 设置西班牙语言代码
    response.setHeader("Content-Language", "es");

    String title = "En Espa&ntilde;ol";
    String docType =
     "<!doctype html public \"-//w3c//dtd html 4.0 " +
     "transitional//en\">\n";
     out.println(docType +
     "<html>\n" +
     "<head><title>" + title + "</title></head>\n" +
     "<body bgcolor=\"#f0f0f0\">\n" +
     "<h1>" + "En Espa&ntilde;ol:" + "</h1>\n" +
     "<h1>" + "&iexcl;Hola Mundo!" + "</h1>\n" +
     "</body></html>");
  }
}
```

特定于区域设置的日期

您可以使用 java.text.DateFormat 类及其静态方法 getDateTimeInstance() 来格式化特定于区域设置的日期和时间。下面的实例演示了如何格式化特定于某个给定的区域设置的日期：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;
import java.text.DateFormat;
import java.util.Date;

public class DateLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 获取客户端的区域设置
    Locale locale = request.getLocale( );
    String date = DateFormat.getDateTimeInstance(
                                  DateFormat.FULL,
                                  DateFormat.SHORT,
                                  locale).format(new Date( ));

    String title = "特定于区域设置的日期";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n" +
      "<h1 align=\"center\">" + date + "</h1>\n" +
      "</body></html>");
  }
}
```

特定于区域设置的货币

您可以使用 java.text.NumberFormat 类及其静态方法 getCurrencyInstance() 来格式化数字（比如 long 类型或 double 类型）为特定于区域设置的货币。下面的实例演示了如何格式化特定于某个给定的区域设置的货币：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;
import java.text.NumberFormat;
import java.util.Date;

public class CurrencyLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 获取客户端的区域设置
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getCurrencyInstance(locale);
    String formattedCurr = nft.format(1000000);

    String title = "特定于区域设置的货币";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n" +
      "<h1 align=\"center\">" + formattedCurr + "</h1>\n" +
      "</body></html>");
  }
}
```

特定于区域设置的百分比

您可以使用 java.text.NumberFormat 类及其静态方法 getPercentInstance() 来格式化特定于区域设置的百分比。下面的实例演示了如何格式化特定于某个给定的区域设置的百分比：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Locale;
import java.text.NumberFormat;
import java.util.Date;

public class PercentageLocale extends HttpServlet{

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException
  {
    // 设置响应内容类型
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    // 获取客户端的区域设置
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getPercentInstance(locale);
    String formattedPerc = nft.format(0.51);

    String title = "特定于区域设置的百分比";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " +
      "transitional//en\">\n";
      out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n" +
      "<h1 align=\"center\">" + formattedPerc + "</h1>\n" +
      "</body></html>");
  }
}
```

JSP 基础

JSP 简介

什么是Java Server Pages?

JSP全称Java Server Pages，是一种动态网页开发技术。它使用JSP标签在HTML网页中插入Java代码。标签通常以<%开头以%>结束。

JSP是一种Java servlet，主要用于实现Java web应用程序的用户界面部分。网页开发者们通过结合HTML代码、XHTML代码、XML元素以及嵌入JSP操作和命令来编写JSP。

JSP通过网页表单获取用户输入数据、访问数据库及其他数据源，然后动态地创建网页。

JSP标签有多种功能，比如访问数据库、记录用户选择信息、访问JavaBeans组件等，还可以在不同的网页中传递控制信息和共享信息。

为什么使用JSP？

JSP程序与CGI程序有着相似的功能，但和CGI程序相比，JSP程序有如下优势：

- 性能更加优越，因为JSP可以直接在HTML网页中动态嵌入元素而不需要单独引用CGI文件。
- 服务器调用的是已经编译好的JSP文件，而不像CGI/Perl那样必须先载入解释器和目标脚本。
- JSP基于Java Servlets API，因此，JSP拥有各种强大的企业级Java API，包括JDBC，JNDI，EJB，JAXP等等。
- JSP页面可以与处理业务逻辑的servlets一起使用，这种模式被Java servlet 模板引擎所支持。

最后，JSP是Java EE不可或缺的一部分，是一个完整的企业级应用平台。这意味着JSP可以用最简单的方式来实现最复杂的应用。

JSP的优势

以下列出了使用JSP带来的其他好处：

- 与ASP相比：JSP有两大优势。首先，动态部分用Java编写，而不是VB或其他MS专用语言，所以更加强大与易用。第二点就是JSP易于移植到非MS平台上。
- 与纯 Servlets相比：JSP可以很方便的编写或者修改HTML网页而不用去面对大量的println语句。
- 与SSI相比：SSI无法使用表单数据、无法进行数据库链接。

- 与JavaScript相比：虽然JavaScript可以在客户端动态生成HTML，但是很难与服务器交互，因此不能提供复杂的服务，比如访问数据库和图像处理等等。
- 与静态HTML相比：静态HTML不包含动态信息。

接下来呢？

我们将会带您一步一步地来搭建JSP运行环境，这需要有一定的Java基础。

如果您还未学过Java，可以先学习我们为您提供的Java教程。

JSP 开发环境搭建

JSP开发环境是您用来开发、测试和运行JSP程序的地方。

本节将会带您搭建JSP开发环境，具体包括以下几个步骤。

配置Java开发工具（JDK）

这一步涉及Java SDK的下载和PATH环境变量的配置。

您可以从Oracle公司的Java页面中下载SDK：Java SE Downloads

Java SDK下载完后，请按照给定的指示来安装和配置SDK。最后，通过设置PATH和JAVA_HOME环境变量来指明包括java和javac的文件夹路径，通常是java_install_dir/bin和java_install_dir。

假如您用的是Windows系统并且SDK的安装目录为C::\jdk1.5.0_20，那么您就需要在C:\autoexec.bat 文件中添加以下两行：

```
set PATH=C:\jdk1.5.0_20\bin;%PATH%
set JAVA_HOME=C:\jdk1.5.0_20
```

或者，在Windows NT/2000/XP下，您可以直接右击我的电脑图标，选择属性，然后高级，然后环境变量，接下来您就可以很方便地设置PATH变量并且确定退出就行了。

在Linux/Unix系统下，如果SDK的安装目录为/usr/local/jdk1.5.0_20并且使用的是C shell，那么您就需要在.cshrc文件中添加以下两行：

```
setenv PATH /usr/local/jdk1.5.0_20/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.5.0_20
```

或者，假如您正在使用类似于Borland JBuilder、Eclipse、IntelliJ IDEA和Sun ONE Studio这样的集成开发环境，可以试着编译并运行一个简单的程序来确定IDE（集成开发环境）是否已经知道 SDK的安装目录。

本步骤你也可以参考本站Java开发环境配置章节的教程。

设置Web服务器：Tomcat

目前，市场上有很多支持JSP和Servlets开发的Web服务器。他们中的一些可以免费下载和使用，Tomcat就是其中之一。

Apache Tomcat是一个开源软件，可作为独立的服务器来运行JSP和Servlets，也可以集成在Apache Web Server中。以下是Tomcat的配置方法：

- 下载最新版本的Tomcat：http://tomcat.apache.org/。
- 下载完安装文件后，将压缩文件解压到一个方便的地方，比如Windows下的C:\apache-tomcat-5.5.29目录或者Linux/Unix下的/usr/local/apache-tomcat-5.5.29目录，然后创建CATALINA_HOME环境变量指向这些目录。

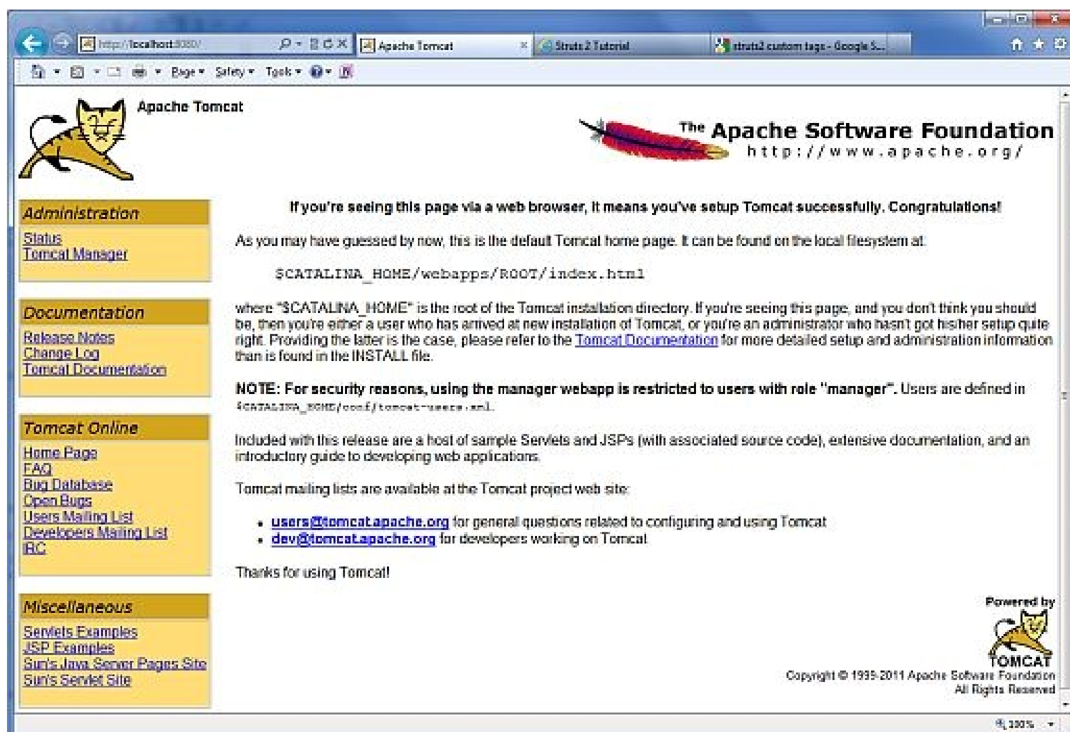
在Windows机器下，Tomcat可以通过执行以下命令来启动：

```
%CATALINA_HOME%\bin\startup.bat
或者
C:\apache-tomcat-5.5.29\bin\startup.bat
```

在Linux/Unix机器下，Tomcat可以通过执行以下命令来启动：

```
$CATALINA_HOME/bin/startup.sh
或者
/usr/local/apache-tomcat-5.5.29/bin/startup.sh
```

成功启动Tomcat后，通过访问http://localhost:8080/便可以使用Tomcat自带的一些web应用了。假如一切顺利的话，您应该能够看到以下的页面：



更多关于配置和运行Tomcat的信息可以在Tomcat提供的文档中找到，或者去Tomcat官网查阅：http://tomcat.apache.org。

在Windows机器下，Tomcat可以通过执行以下命令来停止：

```
%CATALINA_HOME%\bin\shutdown
或者
C:\apache-tomcat-5.5.29\bin\shutdown
```

在Linux/Unix机器下，Tomcat可以通过执行以下命令来停止：

```
$CATALINA_HOME/bin/shutdown.sh
或者
/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

设置CLASSPATH环境变量

由于servlets不是Java SE的一部分，所以您必须标示出servlet类的编译器。

假如您用的是Windows机器，您需要在C:\autoexec.bat文件中添加以下两行：

```
set CATALINA=C:\apache-tomcat-5.5.29
set CLASSPATH=%CATALINA%\common\lib\jsp-api.jar;%CLASSPATH%
```

或者，在Windows NT/2000/XP下，您只要右击我的电脑，选择属性，然后点击高级，然后点击环境变量，接下来便可以设置CLASSPATH变量并且确定退出即可。

在Linux/Unix机器下，假如您使用的是C shell，那么您就需要在.cshrc文件中添加以下两行：

```
setenv CATALINA=/usr/local/apache-tomcat-5.5.29
setenv CLASSPATH $CATALINA/common/lib/jsp-api.jar:$CLASSPATH
```

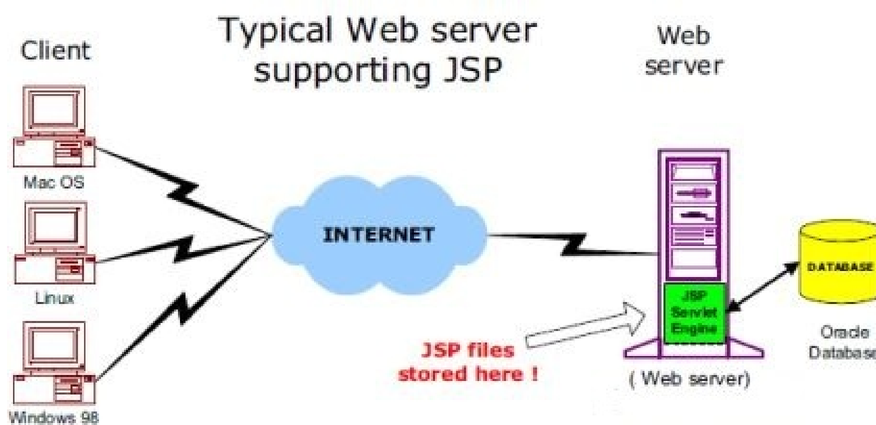
注意：如果您的开发路径是C:\JSPDev (Windows)或者 /usr/JSPDev (Linux/Unix)，那么您就需要将这些路径添加进CLASSPATH变量中。

JSP 结构

网络服务器需要一个JSP引擎，也就是一个容器来处理JSP页面。容器负责截获对JSP页面的请求。本教程使用内嵌JSP容器的Apache来支持JSP开发。

JSP容器与Web服务器协同合作，为JSP的正常运行提供必要的运行环境和其他服务，并且能够正确识别专属于JSP网页的特殊元素。

下图显示了JSP容器和JSP文件在Web应用中所处的位置。

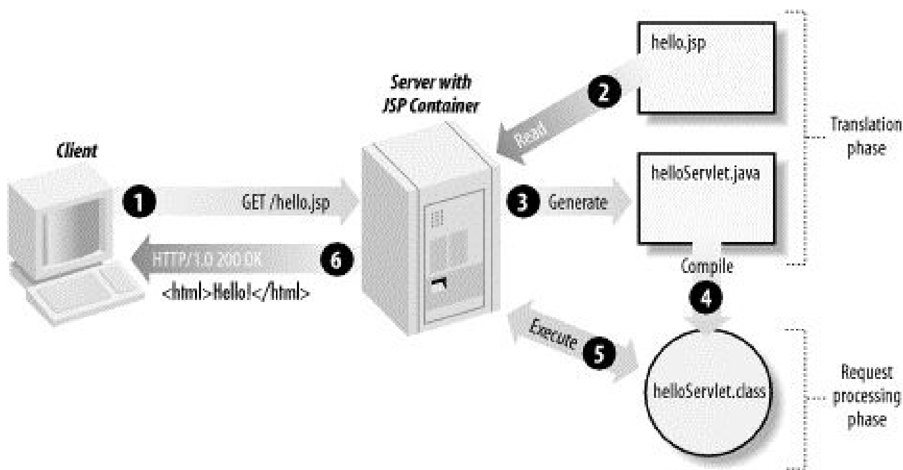


JSP处理

以下步骤表明了Web服务器是如何使用JSP来创建网页的：

- 就像其他普通的网页一样，您的浏览器发送一个HTTP请求给服务器。
- Web服务器识别出这是一个对JSP网页的请求，并且将该请求传递给JSP引擎。通过使用URL或者.jsp文件来完成。
- JSP引擎从磁盘中载入JSP文件，然后将它们转化为servlet。这种转化只是简单地将所有模板文本改用println()语句，并且将所有的JSP元素转化成Java代码。
- JSP引擎将servlet编译成可执行类，并且将原始请求传递给servlet引擎。
- Web服务器的某组件将会调用servlet引擎，然后载入并执行servlet类。在执行过程中，servlet产生HTML格式的输出并将其内嵌于HTTP response中上交给Web服务器。
- Web服务器以静态HTML网页的形式将HTTP response返回到您的浏览器中。
- 最终，Web浏览器处理HTTP response中动态产生的HTML网页，就好像在处理静态网页一样。

以上提及到的步骤可以用下图来表示：



一般情况下，JSP引擎会检查JSP文件对应的servlet是否已经存在，并且检查JSP文件的修改日期是否早于servlet。如果JSP文件的修改日期早于对应的servlet，那么容器就可以确定JSP文件没有被修改过并且servlet有效。这使得整个流程与其他脚本语言（比如PHP）相比要高效快捷一些。

总的来说，JSP网页就是用另一种方式来编写servlet而不用成为Java编程高手。除了解释阶段外，JSP网页几乎可以被当成一个普通的servlet来对待。

JSP 生命周期

理解JSP底层功能的关键就是去理解它们所遵守的生命周期。

JSP生命周期就是从创建到销毁的整个过程，类似于servlet生命周期，区别在于JSP生命周期还包括将JSP文件编译成servlet。

以下是JSP生命周期中所走过的几个阶段：

- 编译阶段：

  servlet容器编译servlet源文件，生成servlet类

- 初始化阶段：

  加载与JSP对应的servlet类，创建其实例，并调用它的初始化方法

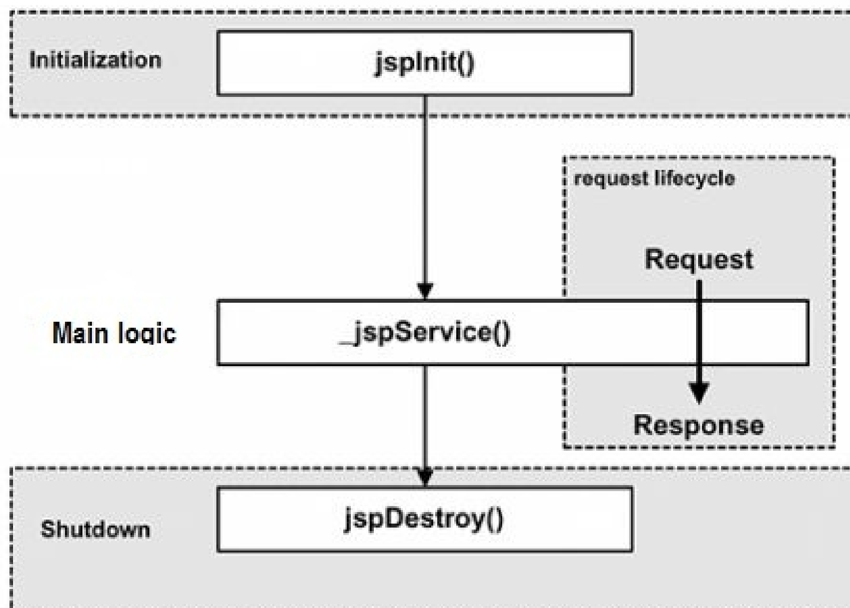
- 执行阶段：

  调用与JSP对应的servlet实例的服务方法

- 销毁阶段：

  调用与JSP对应的servlet实例的销毁方法，然后销毁servlet实例

很明显，JSP生命周期的四个主要阶段和servlet生命周期非常相似，下面给出图示：



JSP编译

当浏览器请求JSP页面时，JSP引擎会首先去检查是否需要编译这个文件。如果这个文件没有被编译过，或者在上次编译后被更改过，则编译这个JSP文件。

编译的过程包括三个步骤：

- 解析JSP文件。
- 将JSP文件转为servlet。
- 编译servlet。

JSP初始化

容器载入JSP文件后，它会在为请求提供任何服务前调用jspInit()方法。如果您需要执行自定义的JSP初始化任务，复写jspInit()方法就行了，就像下面这样：

```
public void jspInit(){
  // 初始化代码
}
```

一般来讲程序只初始化一次，servlet也是如此。通常情况下您可以在jspInit()方法中初始化数据库连接、打开文件和创建查询表。

JSP执行

这一阶段描述了JSP生命周期中一切与请求相关的交互行为，直到被销毁。

当JSP网页完成初始化后，JSP引擎将会调用_jspService()方法。

_jspService()方法需要一个HttpServletRequest对象和一个HttpServletResponse对象作为它的参数，就像下面这样：

```
void _jspService(HttpServletRequest request,
                 HttpServletResponse response)
{
   // 服务端处理代码
}
```

_jspService()方法在每个request中被调用一次并且负责产生与之相对应的response，并且它还负责产生所有7个HTTP方法的回应，比如GET、POST、DELETE等等。

JSP清理

JSP生命周期的销毁阶段描述了当一个JSP网页从容器中被移除时所发生的一切。

jspDestroy()方法在JSP中等价于servlet中的销毁方法。当您需要执行任何清理工作时复写jspDestroy()方法，比如释放数据库连接或者关闭文件夹等等。

jspDestroy()方法的格式如下：

```
public void jspDestroy()
{
   // 清理代码
}
```

实例

JSP生命周期代码实例如下所示：

```
<%@ page contentType="text/html; charset=GB2312" %>
<html><head><title>life.jsp</title></head><body>

<%!
  private int initVar=0;
  private int serviceVar=0;
  private int destroyVar=0;
%>

<%!
  public void jspInit(){
    initVar++;
    System.out.println("jspInit(): JSP被初始化了"+initVar+"次");
  }
  public void jspDestroy(){
    destroyVar++;
    System.out.println("jspDestroy(): JSP被销毁了"+destroyVar+"次");
  }
%>

<%
  serviceVar++;
  System.out.println("_jspService(): JSP共响应了"+serviceVar+"次请求");

  String content1="初始化次数 : "+initVar;
  String content2="响应客户请求次数 : "+serviceVar;
  String content3="销毁次数 : "+destroyVar;
%>

<h1><%=content1 %></h1>
<h1><%=content2 %></h1>
<h1><%=content3 %></h1>

</body></html>
```

JSP 语法

本小节将会简单地介绍一下JSP开发中的基础语法。

脚本程序

脚本程序可以包含任意量的Java语句、变量、方法或表达式，只要它们在脚本语言中是有效的。

脚本程序的语法格式：

```
<% 代码片段 %>
```

或者，您也可以编写与其等价的XML语句，就像下面这样：

```
<jsp:scriptlet>
   代码片段
</jsp:scriptlet>
```

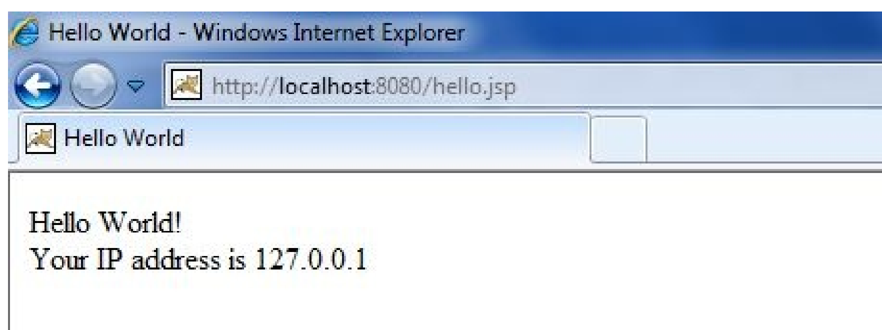
任何文本、HTML标签、JSP元素必须写在脚本程序的外面。

下面给出一个示例，同时也是本教程的第一个JSP示例：

```
<html>
<head><title>Hello World</title></head>
<body>
Hello World!<br/>
<%
out.println("Your IP address is " + request.getRemoteAddr());
%>
</body>
</html>
```

注意：请确保Apache Tomcat已经安装在C:\apache-tomcat-7.0.2目录下并且运行环境已经正确设置。

将以上代码保存在hello.jsp中，然后将它放置在 C:\apache-tomcat-7.0.2\webapps\ROOT目录下，打开浏览器并在地址栏中输入http://localhost:8080/hello.jsp。运行后得到以下结果：



JSP声明

一个声明语句可以声明一个或多个变量、方法，供后面的Java代码使用。在JSP文件中，您必须先声明这些变量和方法然后才能使用它们。

JSP声明的语法格式：

```
<%! declaration; [ declaration; ]+ ... %>
```

或者，您也可以编写与其等价的XML语句，就像下面这样：

```
<jsp:declaration>
   代码片段
</jsp:declaration>
```

程序示例：

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

JSP表达式

一个JSP表达式中包含的脚本语言表达式，先被转化成String，然后插入到表达式出现的地方。

由于表达式的值会被转化成String，所以您可以在一个文本行中使用表达式而不用去管它是否是HTML标签。

表达式元素中可以包含任何符合Java语言规范的表达式，但是不能使用分号来结束表达式。

JSP表达式的语法格式：

```
<%= 表达式 %>
```

同样，您也可以编写与之等价的XML语句：

```
<jsp:expression>
   表达式
</jsp:expression>
```

程序示例：

```
<html>
<head><title>A Comment Test</title></head>
<body>
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
</body>
</html>
```

运行后得到以下结果：

```
Today's date: 11-Sep-2013 21:24:25
```

JSP注释

JSP注释主要有两个作用：为代码作注释以及将某段代码注释掉。

JSP注释的语法格式：

```
<%-- 这里可以填写 JSP 注释 --%>
```

程序示例：

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
<%-- 该部分注释在网页中不会被显示--%>
</body>
</html>
```

运行后得到以下结果：

```
A Test of Comments
```

不同情况下使用注释的语法规则：

语法	描述
<%-- 注释 --%>	JSP注释，注释内容不会被发送至浏览器甚至不会被编译
<!-- 注释 -->	HTML注释，通过浏览器查看网页源代码时可以看见注释内容
<\%	代表静态 <%常量
%\>	代表静态 %> 常量
\'	在属性中使用的单引号
\"	在属性中使用的双引号

JSP指令

JSP指令用来设置与整个JSP页面相关的属性。

JSP指令语法格式：

```
<%@ directive attribute="value" %>
```

这里有三种指令标签：

指令	描述
<%@ page ... %>	定义页面的依赖属性，比如脚本语言、error页面、缓存需求等等
<%@ include ... %>	包含其他文件
<%@ taglib ... %>	引入标签库的定义，可以是自定义标签

JSP行为

JSP行为标签使用XML语法结构来控制servlet引擎。它能够动态插入一个文件，重用JavaBean组件，引导用户去另一个页面，为Java插件产生相关的HTML等等。

行为标签只有一种语法格式，它严格遵守XML标准：

```
<jsp:action_name attribute="value" />
```

行为标签基本上是一些预先就定义好的函数，下表罗列出了一些可用的JSP行为标签：：

语法	描述
jsp:include	用于在当前页面中包含静态或动态资源
jsp:useBean	寻找和初始化一个JavaBean组件
jsp:setProperty	设置 JavaBean组件的值
jsp:getProperty	将 JavaBean组件的值插入到 output中
jsp:forward	从一个JSP文件向另一个文件传递一个包含用户请求的request对象
jsp:plugin	用于在生成的HTML页面中包含Applet和JavaBean对象
jsp:element	动态创建一个XML元素
jsp:attribute	定义动态创建的XML元素的属性
jsp:body	定义动态创建的XML元素的主体
jsp:text	用于封装模板数据

JSP隐含对象

JSP支持九个自动定义的变量，江湖人称隐含对象。这九个隐含对象的简介见下表：

对象	描述
request	**HttpServletRequest**类的实例
response	**HttpServletResponse**类的实例
out	**PrintWriter**类的实例，用于把结果输出至网页上
session	**HttpSession**类的实例
application	**ServletContext**类的实例，与应用上下文有关
config	**ServletConfig**类的实例
pageContext	**PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问
page	类似于Java类中的this关键字
Exception	**Exception**类的对象，代表发生错误的JSP页面中对应的异常对象

控制流语句

JSP提供对Java语言的全面支持。您可以在JSP程序中使用Java API甚至建立Java代码块，包括判断语句和循环语句等等。

判断语句

If…else块，请看下面这个例子：

```
<%! int day = 3; %>
<html>
<head><title>IF...ELSE Example</title></head>
<body>
<% if (day == 1 | day == 7) { %>
      <p> Today is weekend</p>
<% } else { %>
      <p> Today is not weekend</p>
<% } %>
</body>
</html>
```

运行后得到以下结果：

```
Today is not weekend
```

现在来看看switch…case块，与if…else块有很大的不同，它使用out.println()，并且整个都装在脚本程序的标签中，就像下面这样：

```
<%! int day = 3; %>
<html>
<head><title>SWITCH...CASE Example</title></head>
<body>
<%
switch(day) {
case 0:
   out.println("It\'s Sunday.");
   break;
case 1:
   out.println("It\'s Monday.");
   break;
case 2:
   out.println("It\'s Tuesday.");
   break;
case 3:
   out.println("It\'s Wednesday.");
   break;
case 4:
   out.println("It\'s Thursday.");
   break;
case 5:
   out.println("It\'s Friday.");
   break;
default:
   out.println("It's Saturday.");
}
%>
</body>
</html>
```

运行后得出以下结果：

```
It's Wednesday.
```

循环语句

在JSP程序中可以使用Java的三个基本循环类型：for，while，和 do…while。

让我们来看看for循环的例子：

```
<%! int fontSize; %>
<html>
<head><title>FOR LOOP Example</title></head>
<body>
<%for ( fontSize = 1; fontSize <= 3; fontSize++){ %>
   <font color="green" size="<%= fontSize %>">
    JSP Tutorial
   </font><br />
<%}%>
</body>
</html>
```

运行后得到以下结果：

JSP Tutorial
JSP Tutorial
JSP Tutorial

将上例改用while循环来写：

```
<%! int fontSize; %>
<html>
<head><title>WHILE LOOP Example</title></head>
<body>
<%while ( fontSize <= 3){ %>
   <font color="green" size="<%= fontSize %>">
    JSP Tutorial
   </font><br />
<%fontSize++;%>
<%}%>
</body>
</html>
```

运行后得到同样的结果：
JSP Tutorial
JSP Tutorial
JSP Tutorial

JSP运算符

JSP支持所有Java逻辑和算术运算符。

下表罗列出了JSP常见运算符，优先级从高到底：

类别	操作符	结合性
后缀	() [] . (点运算符)	左到右
一元	++ - - ! ~	右到左
可乘性	* / %	左到右
可加性	+ -	左到右
移位	>> >>> <<	左到右
关系	> >= < <=	左到右
相等/不等	== !=	左到右
位与	&	左到右
位异或	^	左到右
位或	|	左到右
逻辑与	&&	左到右
逻辑或	||	左到右
条件判断	?:	右到左
赋值	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

JSP常量

JSP语言定义了以下几个常量：

- Boolean：true and false
- Integer：与Java中的一样
- Floating point：与Java中的一样
- String：以单引号或双引号开始和结束。 " 被转义成 \"，'被转义成 \'， \ 被转义成\
- Null：null

JSP 指令

JSP指令用来设置整个JSP页面相关的属性，如网页的编码方式和脚本语言。

语法格式如下：

```
<%@ directive attribute="value" %>
```

指令可以有很多个属性，它们以键值对的形式存在，并用逗号隔开。

JSP中的三种指令标签：

指令	描述
<%@ page ... %>	定义网页依赖属性，比如脚本语言、error页面、缓存需求等等
<%@ include ... %>	包含其他文件
<%@ taglib ... %>	引入标签库的定义

Page指令

Page指令为容器提供当前页面的使用说明。一个JSP页面可以包含多个page指令。

Page指令的语法格式：

```
<%@ page attribute="value" %>
```

等价的XML格式：

```
<jsp:directive.page attribute="value" />
```

属性

下表列出与Page指令相关的属性：

属性	描述
buffer	指定out对象使用缓冲区的大小
autoFlush	控制out对象的 缓存区
contentType	指定当前JSP页面的MIME类型和字符编码
errorPage	指定当JSP页面发生异常时需要转向的错误处理页面
isErrorPage	指定当前页面是否可以作为另一个JSP页面的错误处理页面
extends	指定servlet从哪一个类继承
import	导入要使用的Java类
info	定义JSP页面的描述信息
isThreadSafe	指定对JSP页面的访问是否为线程安全
language	定义JSP页面所用的脚本语言，默认是Java
session	指定JSP页面是否使用session
isELIgnored	指定是否执行EL表达式
isScriptingEnabled	确定脚本元素能否被使用

Include指令

JSP可以通过include指令来包含其他文件。被包含的文件可以是JSP文件、HTML文件或文本文件。包含的文件就好像是该JSP文件的一部分，会被同时编译执行。

Include指令的语法格式如下：

```
<%@ include file="relative url" %>
```

Include指令中的文件名实际上是一个相对的URL。如果您没有给文件关联一个路径，JSP编译器默认在当前路径下寻找。

等价的XML语法：

```
<jsp:directive.include file="relative url" />
```

Taglib指令

JSP API允许用户自定义标签，一个自定义标签库就是自定义标签的集合。

Taglib指令引入一个自定义标签集合的定义，包括库路径、自定义标签。

Taglib指令的语法：

```
<%@ taglib uri="uri" prefix="prefixOfTag" %>
```

uri属性确定标签库的位置，prefix属性指定标签库的前缀。

等价的XML语法：

```
<jsp:directive.taglib uri="uri" prefix="prefixOfTag" />
```

JSP 动作元素

与JSP指令元素不同的是，JSP动作元素在请求处理阶段起作用。JSP动作元素是用XML语法写成的。

利用JSP动作可以动态地插入文件、重用JavaBean组件、把用户重定向到另外的页面、为Java插件生成HTML代码。

动作元素只有一种语法，它符合XML标准：

```
<jsp:action_name attribute="value" />
```

动作元素基本上都是预定义的函数，JSP规范定义了一系列的标准动作，它用JSP作为前缀，可用的标准动作元素如下：

语法	描述
jsp:include	在页面被请求的时候引入一个文件。
jsp:useBean	寻找或者实例化一个JavaBean。
jsp:setProperty	设置JavaBean的属性。
jsp:getProperty	输出某个JavaBean的属性。
jsp:forward	把请求转到一个新的页面。
jsp:plugin	根据浏览器类型为Java插件生成OBJECT或EMBED标记。
jsp:element	定义动态XML元素
jsp:attribute	设置动态定义的XML元素属性。
jsp:body	设置动态定义的XML元素内容。
jsp:text	在JSP页面和文档中使用写入文本的模板

常见的属性

所有的动作要素都有两个属性：id属性和scope属性。

- **id**属性：

  id属性是动作元素的唯一标识，可以在JSP页面中引用。动作元素创建的id值可以通过PageContext来调用。

- **scope**属性：

> 该属性用于识别动作元素的生命周期。 id属性和scope属性有直接关系，scope属性定义了相关联id对象的寿命。 scope属性有四个可能的值： (a) page, (b)request, (c)session, 和 (d) application。

<jsp:include>动作元素

<jsp:include>动作元素用来包含静态和动态的文件。该动作把指定文件插入正在生成的页面。语法格式如下：

```
<jsp:include page="relative URL" flush="true" />
```

前面已经介绍过include指令，它是在JSP文件被转换成Servlet的时候引入文件，而这里的jsp:include动作不同，插入文件的时间是在页面被请求的时候。

以下是include动作相关的属性列表。

属性	描述
page	包含在页面中的相对URL地址。
flush	布尔属性，定义在包含资源前是否刷新缓存区。

实例

以下我们定义了两个文件date.jsp和main.jsp，代码如下所示：

date.jsp文件代码：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
The include action Example
Today's date: 12-Sep-2013 14:54:22
```

<jsp:useBean>动作元素

jsp:useBean动作用来装载一个将在JSP页面中使用的JavaBean。

这个功能非常有用，因为它使得我们既可以发挥Java组件重用的优势，同时也避免了损失JSP区别于Servlet的方便性。

jsp:useBean动作最简单的语法为：

```
<jsp:useBean id="name" class="package.class" />
```

在类载入后，我们既可以通过 jsp:setProperty 和 jsp:getProperty 动作来修改和检索bean的属性。

以下是useBean动作相关的属性列表。

属性	描述
class	指定Bean的完整包名。
type	指定将引用该对象变量的类型。
beanName	通过 java.beans.Beans 的 instantiate() 方法指定Bean的名字。

在给出具体实例前，让我们先来看下 jsp:setProperty 和 jsp:getProperty 动作元素：

<jsp:setProperty>动作元素

jsp:setProperty用来设置已经实例化的Bean对象的属性，有两种用法。首先，你可以在jsp:useBean元素的外面（后面）使用jsp:setProperty，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" .../>
```

此时，不管jsp:useBean是找到了一个现有的Bean，还是新创建了一个Bean实例，jsp:setProperty都会执行。第二种用法是把jsp:setProperty放入jsp:useBean元素的内部，如下所示：

```
<jsp:useBean id="myName" ... >
...
   <jsp:setProperty name="myName" property="someProperty" .../>
</jsp:useBean>
```

此时，jsp:setProperty只有在新建Bean实例时才会执行，如果是使用现有实例则不执行jsp:setProperty。

jsp:setproperty动作有下面四个属性,如下表：

属性	描述
name	name属性是必需的。它表示要设置属性的是哪个Bean。
property	property属性是必需的。它表示要设置哪个属性。有一个特殊用法：如果property的值是"*"，表示所有名字和Bean属性名字匹配的请求参数都将被传递给相应的属性set方法。
value	value 属性是可选的。该属性用来指定Bean属性的值。字符串数据会在目标类中通过标准的valueOf方法自动转换成数字、boolean、Boolean、 byte、Byte、char、Character。例如，boolean和Boolean类型的属性值（比如"true"）通过 Boolean.valueOf转换，int和Integer类型的属性值（比如"42"）通过Integer.valueOf转换。 value和param不能同时使用，但可以使用其中任意一个。
param	param 是可选的。它指定用哪个请求参数作为Bean属性的值。如果当前请求没有参数，则什么事情也不做，系统不会把null传递给Bean属性的set方法。因此，你可以让Bean自己提供默认属性值，只有当请求参数明确指定了新值时才修改默认属性值。

<jsp:getProperty>动作元素

jsp:getProperty动作提取指定Bean属性的值，转换成字符串，然后输出。语法格式如下：

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name="myName" property="someProperty" .../>
```

下表是与getProperty相关联的属性：

属性	描述
name	要检索的Bean属性名称。Bean必须已定义。
property	表示要提取Bean属性的值

实例

以下实例我们使用了Bean:

```
/* 文件: TestBean.java */
package action;

public class TestBean {
   private String message = "No message specified";

   public String getMessage() {
      return(message);
   }
   public void setMessage(String message) {
      this.message = message;
   }
}
```

编译以上实例并生成 TestBean.class 文件，将该文件拷贝至服务器正式存放Java类的目录下，而不是保留给修改后能够自动装载的类的目录(如：C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action目录中，CLASSPATH 变量必须包含该路径。)。例如，对于Java Web Server来说，Bean和所有Bean用到的类都应该放入classes目录，或者封装进jar文件后放入lib目录，但不应该放到servlets 下。 下面是一个很简单的例子，它的功能是装载一个Bean，然后设置/读取它的message属性。

现在让我们在main.jsp文件中调用该Bean:

```
 <html>
<head>
<title>Using JavaBeans in JSP</title>
</head>
<body>
<center>
<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"
                    property="message"
                    value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>
</body>
</html>
```

执行以上文件，输出如下所示：

```
Using JavaBeans in JSP
Got message....
Hello JSP...
```

<jsp:forward> 动作元素

jsp:forward动作把请求转到另外的页面。jsp:forward标记只有一个属性page。语法格式如下所示：

```
<jsp:forward page="Relative URL" />
```

以下是forward相关联的属性：

属性	描述
page	page属性包含的是一个相对URL。page的值既可以直接给出，也可以在请求的时候动态计算，可以是一个JSP页面或者一个 Java Servlet.

实例

以下实例我们使用了两个文件，分别是： date.jps 和 main.jsp。

date.js文件代码如下：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The forward Action Example</title>
</head>
<body>
<center>
<h2>The forward action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
Today's date: 12-Sep-2010 14:54:22
```

<jsp:plugin>动作元素

jsp:plugin动作用来根据浏览器的类型，插入通过Java插件 运行Java Applet所必需的OBJECT或EMBED元素。

如果需要的插件不存在，它会下载插件，然后执行Java组件。 Java组件可以是一个applet或一个JavaBean。

plugin动作有多个对应HTML元素的属性用于格式化Java 组件。param元素可用于向Applet 或 Bean 传递参数。

以下是使用plugin 动作元素的典型实例:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                           width="60" height="80">
   <jsp:param name="fontcolor" value="red" />
   <jsp:param name="background" value="black" />

   <jsp:fallback>
      Unable to initialize Java Plugin
   </jsp:fallback>

</jsp:plugin>
```

如果你有兴趣可以尝试使用applet来测试jsp:plugin动作元素，<fallback>元素是一个新元素，在组件出现故障的错误是发送给用户错误信息。

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素动态定义XML元素。动态是非常重要的，这就意味着XML元素在编译时是动态生成的而非静态。

以下实例动态定义了XML元素：

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<jsp:element name="xmlElement">
<jsp:attribute name="xmlElementAttr">
   Value for the attribute
</jsp:attribute>
<jsp:body>
   Body for XML element
</jsp:body>
</jsp:element>
</body>
</html>
```

执行时生成HTML代码如下：

```
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<xmlElement xmlElementAttr="Value for the attribute">
   Body for XML element
</xmlElement>
</body>
</html>
```

<jsp:text>动作元素

<jsp:text>动作元素允许在JSP页面和文档中使用写入文本的模板，语法格式如下：

```
<jsp:text>Template data</jsp:text>
```

以上文本模板不能包含其他元素，只能只能包含文本和EL表达式（注：EL表达式将在后续章节中介绍）。请注意，在XML文件中，您不能使用表达式如 ${whatever > 0}，因为>符号是非法的。 你可以使用 ${whatever gt 0}表达式或者嵌入在一个CDATA部分的值。

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

如果你需要在 XHTML 中声明 DOCTYPE,必须使用到<jsp:text>动作元素，实例如下：

```
<jsp:text><![CDATA[<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">]]>
</jsp:text>
<head><title>jsp:text action</title></head>
<body>

<books><book><jsp:text>
    Welcome to JSP Programming
</jsp:text></book></books>

</body>
</html>
```

你可以对以上实例尝试使用<jsp:text>及不使用该动作元素执行结果的区别。

</p   jsp:setproperty动作有下面四个属性,如下表：<>VM1563:1 Resource interpreted as Image but transferred with MIME type text/html: "http://googleads.g.doubleclick.net/pagead/adview?ai=CYd_viH8rVOrQGoGt8gW1-o…dHQJkV0teBb6uOUXPCSK1MkoVQh3X4TKABpy_58Kajo6JdKAGIQ&sigh=pcHBvVQp9GM&vis=1". ads?client=ca-pub-5751451760833794&format=160x600&output=html&h=600&slotname=4106274865&adk=4689099…:1

JSP 动作元素

与JSP指令元素不同的是，JSP动作元素在请求处理阶段起作用。JSP动作元素是用XML语法写成的。

利用JSP动作可以动态地插入文件、重用JavaBean组件、把用户重定向到另外的页面、为Java插件生成HTML代码。

动作元素只有一种语法，它符合XML标准：

```
<jsp:action_name attribute="value" />
```

动作元素基本上都是预定义的函数，JSP规范定义了一系列的标准动作，它用JSP作为前缀，可用的标准动作元素如下：

语法	描述
jsp:include	在页面被请求的时候引入一个文件。
jsp:useBean	寻找或者实例化一个JavaBean。
jsp:setProperty	设置JavaBean的属性。
jsp:getProperty	输出某个JavaBean的属性。
jsp:forward	把请求转到一个新的页面。
jsp:plugin	根据浏览器类型为Java插件生成OBJECT或EMBED标记。
jsp:element	定义动态XML元素
jsp:attribute	设置动态定义的XML元素属性。
jsp:body	设置动态定义的XML元素内容。
jsp:text	在JSP页面和文档中使用写入文本的模板

常见的属性

所有的动作要素都有两个属性：id属性和scope属性。

- **id**属性：

  id属性是动作元素的唯一标识，可以在JSP页面中引用。动作元素创建的id值可以通过PageContext来调用。

- **scope**属性：

该属性用于识别动作元素的生命周期。 id属性和scope属性有直接关系，scope属性定义了相关联id对象的寿命。 scope属性有四个可能的值： (a) page, (b)request, (c)session, 和 (d) application。

<jsp:include>动作元素

<jsp:include>动作元素用来包含静态和动态的文件。该动作把指定文件插入正在生成的页面。语法格式如下：

```
<jsp:include page="relative URL" flush="true" />
```

前面已经介绍过include指令，它是在JSP文件被转换成Servlet的时候引入文件，而这里的jsp:include动作不同，插入文件的时间是在页面被请求的时候。

以下是include动作相关的属性列表。

属性	描述
page	包含在页面中的相对URL地址。
flush	布尔属性，定义在包含资源前是否刷新缓存区。

实例

以下我们定义了两个文件date.jsp和main.jsp，代码如下所示：

date.jsp文件代码：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The include Action Example</title>
</head>
<body>
<center>
<h2>The include action Example</h2>
<jsp:include page="date.jsp" flush="true" />
</center>
</body>
</html>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
The include action Example
Today's date: 12-Sep-2013 14:54:22
```

<jsp:useBean>动作元素

jsp:useBean动作用来装载一个将在JSP页面中使用的JavaBean。

这个功能非常有用，因为它使得我们既可以发挥Java组件重用的优势，同时也避免了损失JSP区别于Servlet的方便性。

jsp:useBean动作最简单的语法为：

```
<jsp:useBean id="name" class="package.class" />
```

在类载入后，我们既可以通过 jsp:setProperty 和 jsp:getProperty 动作来修改和检索bean的属性。

以下是useBean动作相关的属性列表。

属性	描述
class	指定Bean的完整包名。
type	指定将引用该对象变量的类型。
beanName	通过 java.beans.Beans 的 instantiate() 方法指定Bean的名字。

在给出具体实例前，让我们先来看下 jsp:setProperty 和 jsp:getProperty 动作元素：

<jsp:setProperty>动作元素

jsp:setProperty用来设置已经实例化的Bean对象的属性，有两种用法。首先，你可以在jsp:useBean元素的外面（后面）使用jsp:setProperty，如下所示：

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" .../>
```

此时，不管jsp:useBean是找到了一个现有的Bean，还是新创建了一个Bean实例，jsp:setProperty都会执行。第二种用法是把jsp:setProperty放入jsp:useBean元素的内部，如下所示：

```
<jsp:useBean id="myName" ... >
...
   <jsp:setProperty name="myName" property="someProperty" .../>
</jsp:useBean>
```

此时，jsp:setProperty只有在新建Bean实例时才会执行，如果是使用现有实例则不执行jsp:setProperty。

jsp:setproperty动作有下面四个属性,如下表：

属性	描述
name	name属性是必需的。它表示要设置属性的是哪个Bean。
property	property属性是必需的。它表示要设置哪个属性。有一个特殊用法：如果property的值是"*"，表示所有名字和Bean属性名字匹配的请求参数都将被传递给相应的属性set方法。
value	value 属性是可选的。该属性用来指定Bean属性的值。字符串数据会在目标类中通过标准的valueOf方法自动转换成数字、boolean、Boolean、 byte、Byte、char、Character。例如，boolean和Boolean类型的属性值（比如"true"）通过 Boolean.valueOf转换，int和Integer类型的属性值（比如"42"）通过Integer.valueOf转换。 value和param不能同时使用，但可以使用其中任意一个。
param	param 是可选的。它指定用哪个请求参数作为Bean属性的值。如果当前请求没有参数，则什么事情也不做，系统不会把null传递给Bean属性的set方法。因此，你可以让Bean自己提供默认属性值，只有当请求参数明确指定了新值时才修改默认属性值。

<jsp:getProperty>动作元素

jsp:getProperty动作提取指定Bean属性的值，转换成字符串，然后输出。语法格式如下：

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name="myName" property="someProperty" .../>
```

下表是与getProperty相关联的属性：

属性	描述
name	要检索的Bean属性名称。Bean必须已定义。
property	表示要提取Bean属性的值

实例

以下实例我们使用了Bean:

```
/* 文件: TestBean.java */
package action;

public class TestBean {
   private String message = "No message specified";

   public String getMessage() {
      return(message);
   }
   public void setMessage(String message) {
      this.message = message;
   }
}
```

编译以上实例并生成 TestBean.class 文件，将该文件拷贝至服务器正式存放Java类的目录下，而不是保留给修改后能够自动装载的类的目录(如：C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action目录中，CLASSPATH 变量必须包含该路径。)。例如，对于Java Web Server来说，Bean和所有Bean用到的类都应该放入classes目录，或者封装进jar文件后放入lib目录，但不应该放到servlets 下。 下面是一个很简单的例子，它的功能是装载一个Bean，然后设置/读取它的message属性。

现在让我们在main.jsp文件中调用该Bean:

```
 <html>
<head>
<title>Using JavaBeans in JSP</title>
</head>
<body>
<center>
<h2>Using JavaBeans in JSP</h2>

<jsp:useBean id="test" class="action.TestBean" />

<jsp:setProperty name="test"
                    property="message"
                    value="Hello JSP..." />

<p>Got message....</p>

<jsp:getProperty name="test" property="message" />

</center>
</body>
</html>
```

执行以上文件，输出如下所示：

```
Using JavaBeans in JSP
Got message....
Hello JSP...
```

<jsp:forward> 动作元素

jsp:forward动作把请求转到另外的页面。jsp:forward标记只有一个属性page。语法格式如下所示：

```
<jsp:forward page="Relative URL" />
```

以下是forward相关联的属性：

属性	描述
page	page属性包含的是一个相对URL。page的值既可以直接给出，也可以在请求的时候动态计算，可以是一个JSP页面或者一个 Java Servlet.

实例

以下实例我们使用了两个文件，分别是： date.jps 和 main.jsp。

date.js文件代码如下：

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp文件代码：

```
<html>
<head>
<title>The forward Action Example</title>
</head>
<body>
<center>
<h2>The forward action Example</h2>
<jsp:forward page="date.jsp" />
</center>
</body>
```

现在将以上两个文件放在服务器的根目录下，访问main.jsp文件。显示结果如下：

```
Today's date: 12-Sep-2010 14:54:22
```

<jsp:plugin>动作元素

jsp:plugin动作用来根据浏览器的类型，插入通过Java插件 运行Java Applet所必需的OBJECT或EMBED元素。

如果需要的插件不存在，它会下载插件，然后执行Java组件。 Java组件可以是一个applet或一个JavaBean。

plugin动作有多个对应HTML元素的属性用于格式化Java 组件。param元素可用于向Applet 或 Bean 传递参数。

以下是使用plugin 动作元素的典型实例:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class"
                           width="60" height="80">
   <jsp:param name="fontcolor" value="red" />
   <jsp:param name="background" value="black" />

   <jsp:fallback>
      Unable to initialize Java Plugin
   </jsp:fallback>

</jsp:plugin>
```

如果你有兴趣可以尝试使用applet来测试jsp:plugin动作元素，<fallback>元素是一个新元素，在组件出现故障的错误是发送给用户错误信息。

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素

<jsp:element> 、 <jsp:attribute>、 <jsp:body>动作元素动态定义XML元素。动态是非常重要的，这就意味着XML元素在编译时是动态生成的而非静态。

以下实例动态定义了XML元素：

```
<%@page language="java" contentType="text/html"%>
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<jsp:element name="xmlElement">
<jsp:attribute name="xmlElementAttr">
   Value for the attribute
</jsp:attribute>
<jsp:body>
   Body for XML element
</jsp:body>
</jsp:element>
</body>
</html>
```

执行时生成HTML代码如下：

```
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:jsp="http://java.sun.com/JSP/Page">

<head><title>Generate XML Element</title></head>
<body>
<xmlElement xmlElementAttr="Value for the attribute">
   Body for XML element
</xmlElement>
</body>
</html>
```

<jsp:text>动作元素

<jsp:text>动作元素允许在JSP页面和文档中使用写入文本的模板，语法格式如下：

```
<jsp:text>Template data</jsp:text>
```

以上文本模板不能包含其他元素，只能只能包含文本和EL表达式（注：EL表达式将在后续章节中介绍）。请注意，在XML文件中，您不能使用表达式如 ${whatever > 0}，因为>符号是非法的。 你可以使用 ${whatever gt 0}表达式或者嵌入在一个CDATA部分的值。

```
<jsp:text><![CDATA[<br>]]></jsp:text>
```

如果你需要在 XHTML 中声明 DOCTYPE,必须使用到<jsp:text>动作元素，实例如下：

```
<jsp:text><![CDATA[<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">]]>
</jsp:text>
<head><title>jsp:text action</title></head>
<body>

<books><book><jsp:text>
    Welcome to JSP Programming
</jsp:text></book></books>

</body>
</html>
```

你可以对以上实例尝试使用<jsp:text>及不使用该动作元素执行结果的区别。

JSP 隐含对象

JSP隐含对象是JSP容器为每个页面提供的Java对象，开发者可以直接使用它们而不用显式声明。JSP隐含对象也被称为预定义变量。

JSP所支持的九大隐含对象：

对象	描述
request	**HttpServletRequest**类的实例
response	**HttpServletResponse**类的实例
out	**PrintWriter**类的实例，用于把结果输出至网页上
session	**HttpSession**类的实例
application	**ServletContext**类的实例，与应用上下午有关
config	**ServletConfig**类的实例
pageContext	**PageContext**类的实例，提供对JSP页面所有对象以及命名空间的访问
page	类似于Java类中的this关键字
Exception	**Exception**类的对象，代表发生错误的JSP页面中对应的异常对象

request对象

request对象是javax.servlet.http.HttpServletRequest 类的实例。每当客户端请求一个JSP页面时，JSP引擎就会制造一个新的request对象来代表这个请求。

request对象提供了一系列方法来获取HTTP头信息，cookies，HTTP方法等等。

response对象

response对象是javax.servlet.http.HttpServletResponse类的实例。当服务器创建request对象时会同时创建用于响应这个客户端的response对象。

response对象也定义了处理HTTP头模块的接口。通过这个对象，开发者们可以添加新的cookies，时间戳，HTTP状态码等等。

out对象

out对象是 javax.servlet.jsp.JspWriter 类的实例，用来在response对象中写入内容。

最初的JspWriter类对象根据页面是否有缓存来进行不同的实例化操作。可以在page指令中使用buffered='false'属性来轻松关闭缓存。

JspWriter类包含了大部分java.io.PrintWriter类中的方法。不过，JspWriter新增了一些专为处理缓存而设计的方法。还有就是，JspWriter类会抛出IOExceptions异常，而PrintWriter不会。

下表列出了我们将会用来输出boolean，char，int，double，Srtring，object等类型数据的重要方法：

方法	描述
out.print(dataType dt)	输出Type类型的值
out.println(dataType dt)	输出Type类型的值然后换行
out.flush()	刷新输出流

session对象

session对象是 javax.servlet.http.HttpSession 类的实例。和Java Servlets中的session对象有一样的行为。

session对象用来跟踪在各个客户端请求间的会话。

application对象

application对象直接包装了servlet的ServletContext类的对象，是javax.servlet.ServletContext类的实例。

这个对象在JSP页面的整个生命周期中都代表着这个JSP页面。这个对象在JSP页面初始化时被创建，随着jspDestroy()方法的调用而被移除。

通过向application中添加属性，则所有组成您web应用的JSP文件都能访问到这些属性。

config对象

config对象是 javax.servlet.ServletConfig 类的实例，直接包装了servlet的ServletConfig类的对象。

这个对象允许开发者访问Servlet或者JSP引擎的初始化参数，比如文件路径等。

以下是config对象的使用方法，不是很重要，所以不常用：

```
config.getServletName();
```

它返回包含在<servlet-name>元素中的servlet名字，注意，<servlet-name>元素在 WEB-INF\web.xml 文件中定义。

pageContext 对象

pageContext对象是javax.servlet.jsp.PageContext 类的实例，用来代表整个JSP页面。

这个对象主要用来访问页面信息，同时过滤掉大部分实现细节。

这个对象存储了request对象和response对象的引用。application对象，config对象，session对象，out对象可以通过访问这个对象的属性来导出。

pageContext对象也包含了传给JSP页面的指令信息，包括缓存信息，ErrorPage URL,页面scope等。

PageContext类定义了一些字段，包括PAGE_SCOPE，REQUEST_SCOPE，SESSION_SCOPE， APPLICATION_SCOPE。它也提供了40余种方法，有一半继承自javax.servlet.jsp.JspContext 类。

其中一个重要的方法就是removeArribute()，它可接受一个或两个参数。比如，pageContext.removeArribute("attrName")移除四个scope中相关属性，但是下面这种方法只移除特定scope中的相关属性：

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

page 对象

这个对象就是页面实例的引用。它可以被看做是整个JSP页面的代表。

page 对象就是this对象的同义词。

exception 对象

exception 对象包装了从先前页面中抛出的异常信息。它通常被用来产生对出错条件的适当响应。

JSP 客户端请求

当浏览器请求一个网页时，它会向网络服务器发送一系列不能被直接读取的信息，因为这些信息是作为HTTP信息头的一部分来传送的。您可以查阅HTTP协议来获得更多的信息。

下表列出了浏览器端信息头的一些重要内容，在以后的网络编程中将会经常见到这些信息：

信息	描述
Accept	指定浏览器或其他客户端可以处理的MIME类型。它的值通常为 **image/png** 或 **image/jpeg**
Accept-Charset	指定浏览器要使用的字符集。比如 ISO-8859-1
Accept-Encoding	指定编码类型。它的值通常为 **gzip** 或**compress**
Accept-Language	指定客户端首选语言，servlet会优先返回以当前语言构成的结果集，如果servlet支持这种语言的话。比如 en，en-us，ru等等
Authorization	在访问受密码保护的网页时识别不同的用户
Connection	表明客户端是否可以处理HTTP持久连接。持久连接允许客户端或浏览器在一个请求中获取多个文件。**Keep-Alive** 表示启用持久连接
Content-Length	仅适用于POST请求，表示 POST 数据的字节数
Cookie	返回先前发送给浏览器的cookies至服务器
Host	指出原始URL中的主机名和端口号
If-Modified-Since	表明只有当网页在指定的日期被修改后客户端才需要这个网页。 服务器发送304码给客户端，表示没有更新的资源
If-Unmodified-Since	与If-Modified-Since相反， 只有文档在指定日期后仍未被修改过，操作才会成功
Referer	标志着所引用页面的URL。比如，如果你在页面1，然后点了个链接至页面2，那么页面1的URL就会包含在浏览器请求页面2的信息头中
User-Agent	用来区分不同浏览器或客户端发送的请求，并对不同类型的浏览器返回不同的内容

HttpServletRequest类

request对象是javax.servlet.http.HttpServletRequest类的实例。每当客户端请求一个页面时，JSP引擎就会产生一个新的对象来代表这个请求。

request对象提供了一系列方法来获取HTTP信息头，包括表单数据，cookies，HTTP方法等等。

接下来将会介绍一些在JSP编程中常用的获取HTTP信息头的方法。详细内容请见下表：

方法	描述
Cookie[] getCookies()	返回客户端所有的Cookie的数组
Enumeration getAttributeNames()	返回request对象的所有属性名称的集合
Enumeration getHeaderNames()	返回所有HTTP头的名称集合
Enumeration getParameterNames()	返回请求中所有参数的集合
HttpSession getSession()	返回request对应的session对象，如果没有，则创建一个
HttpSession getSession(boolean create)	返回request对应的session对象，如果没有并且参数create为true，则返回一个新的session对象
Locale getLocale()	返回当前页的Locale对象，可以在response中设置
Object getAttribute(String name)	返回名称为name的属性值，如果不存在则返回null。
ServletInputStream getInputStream()	返回请求的输入流
String getAuthType()	返回认证方案的名称，用来保护servlet，比如 "BASIC" 或者 "SSL" 或 null 如果 JSP没设置保护措施
String getCharacterEncoding()	返回request的字符编码集名称
String getContentType()	返回request主体的MIME类型，若未知则返回null
String getContextPath()	返回request URI中指明的上下文路径
String getHeader(String name)	返回name指定的信息头
String getMethod()	返回此request中的HTTP方法，比如 GET,，POST，或PUT
String getParameter(String name)	返回此request中name指定的参数，若不存在则返回null
String getPathInfo()	返回任何额外的与此request URL相关的路径
String getProtocol()	返回此request所使用的协议名和版本
String getQueryString()	返回此 request URL包含的查询字符串
String getRemoteAddr()	返回客户端的IP地址

String getRemoteHost()	返回客户端的完整名称
String getRemoteUser()	返回客户端通过登录认证的用户，若用户未认证则返回null
String getRequestURI()	返回request的URI
String getRequestedSessionId()	返回request指定的session ID
String getServletPath()	返回所请求的servlet路径
String[] getParameterValues(String name)	返回指定名称的参数的所有值，若不存在则返回null
boolean isSecure()	返回request是否使用了加密通道，比如HTTPS
int getContentLength()	返回request主体所包含的字节数，若未知的返回-1
int getIntHeader(String name)	返回指定名称的request信息头的值
int getServerPort()	返回服务器端口号

HTTP信息头示例

在这个例子中，我们会使用HttpServletRequest类的getHeaderNames()方法来读取HTTP信息头。这个方法以枚举的形式返回当前HTTP请求的头信息。

获取Enumeration对象后，用标准的方式来遍历Enumeration对象，用hasMoreElements()方法来确定什么时候停止，用nextElement()方法来获得每个参数的名字。

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>HTTP Header Request Example</title>
</head>
<body>
<center>
<h2>HTTP Header Request Example</h2>
<table width="100%" border="1" align="center">
<tr bgcolor="#949494">
<th>Header Name</th><th>Header Value(s)</th>
</tr>
<%
   Enumeration headerNames = request.getHeaderNames();
   while(headerNames.hasMoreElements()) {
      String paramName = (String)headerNames.nextElement();
      out.print("<tr><td>" + paramName + "</td>\n");
      String paramValue = request.getHeader(paramName);
      out.println("<td> " + paramValue + "</td></tr>\n");
   }
%>
</table>
</center>
</body>
</html>
```

访问main.jsp，将会得到以下结果：

```
<h1>HTTP Header Request Example</h1>

<table width="100%" border="1" align="center">

<tbody>

<tr><th>Header Name</th><th>Header Value(s)</th></tr>

<tr><td>accept</td><td>*/*</td></tr>

<tr><td>accept-language</td><td>en-us</td></tr>

<tr><td>user-agent</td><td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0

<tr><td>accept-encoding</td><td>gzip, deflate</td></tr>

<tr><td>host</td><td>localhost:8080</td></tr>

<tr><td>connection</td><td>Keep-Alive</td></tr>

<tr><td>cache-control</td><td>no-cache</td></tr>

</tbody>

</table>
```

您可以在上面代码中尝试HttpServletRequest类的其它方法。

JSP 服务器响应

Response响应对象主要将JSP容器处理后的结果传回到客户端。可以通过response变量设置HTTP的状态和向客户端发送数据，如Cookie、HTTP文件头信息等。

一个典型的响应看起来就像下面这样：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包含HTTP版本信息，比如HTTP/1.1，一个状态码，比如200，还有一个非常短的信息对应着状态码，比如OK。

下表摘要出了HTTP1.1响应头中最有用的部分，在网络编程中您将会经常见到它们：

响应头	描述
Allow	指定服务器支持的request方法（GET，POST等等）
Cache-Control	指定响应文档能够被安全缓存的情况。通常取值为 **public**，**private** 或**no-cache** 等等。 Public意味着文档可缓存，Private意味着文档只为单用户服务并且只能使用私有缓存。No-cache 意味着文档不被缓存。
Connection	命令浏览器是否要使用持久的HTTP连接。**close**值** 命令浏览器不使用持久HTTP连接，而keep-alive 意味着使用持久化连接。
Content-Disposition	让浏览器要求用户将响应以给定的名称存储在磁盘中
Content-Encoding	指定传输时页面的编码规则
Content-Language	表述文档所使用的语言，比如en， en-us,，ru等等
Content-Length	表明响应的字节数。只有在浏览器使用持久化 (keep-alive) HTTP 连接时才有用
Content-Type	表明文档使用的MIME类型
Expires	指明啥时候过期并从缓存中移除
Last-Modified	指明文档最后修改时间。客户端可以 缓存文档并且在后续的请求中提供一个 **If-Modified-Since**请求头
Location	在300秒内，包含所有的有一个状态码的响应地址，浏览器会自动重连然后检索新文档
Refresh	指明浏览器每隔多久请求更新一次页面。
Retry-After	与503 (Service Unavailable)一起使用来告诉用户多久后请求将会得到响应
Set-Cookie	指明当前页面对应的cookie

HttpServletResponse类

response对象是javax.servlet.http.HttpServletRequest类的一个实例。就像服务器会创建request对象一样，它也会创建一个客户端响应。

response对象定义了处理创建HTTP信息头的接口。通过使用这个对象，开发者们可以添加新的cookie或时间戳，还有HTTP状态吗等等。

下表列出了用来设置HTTP响应头的方法，这些方法由HttpServletResponse 类提供：

方法	描述
String	

encodeRedirectURL(String url)	对sendRedirect()方法使用的URL进行编码
String encodeURL(String url)	将URL编码，回传包含Session ID的URL
boolean containsHeader(String name)	返回指定的响应头是否存在
boolean isCommitted()	返回响应是否已经提交到客户端
void addCookie(Cookie cookie)	添加指定的cookie至响应中
void addDateHeader(String name, long date)	添加指定名称的响应头和日期值
void addHeader(String name, String value)	添加指定名称的响应头和值
void addIntHeader(String name, int value)	添加指定名称的响应头和int值
void flushBuffer()	将任何缓存中的内容写入客户端
void reset()	清除任何缓存中的任何数据，包括状态码和各种响应头
void resetBuffer()	清除基本的缓存数据，不包括响应头和状态码
void sendError(int sc)	使用指定的状态码向客户端发送一个出错响应，然后清除缓存
void sendError(int sc, String msg)	使用指定的状态码和消息向客户端发送一个出错响应
void sendRedirect(String location)	使用指定的URL向客户端发送一个临时的间接响应
void setBufferSize(int size)	设置响应体的缓存区大小
void setCharacterEncoding(String charset)	指定响应的编码集（MIME字符集），例如UTF-8
void setContentLength(int len)	指定HTTP servlets中响应的内容的长度，此方法用来设置 HTTP Content-Length 信息头
void setContentType(String type)	设置响应的内容的类型，如果响应还未被提交的话
void setDateHeader(String name, long date)	使用指定名称和值设置响应头的名称和内容
void setHeader(String name, String value)	使用指定名称和值设置响应头的名称和内容
void setIntHeader(String name, int value)	使用指定名称和值设置响应头的名称和内容
void setLocale(Locale loc)	设置响应的语言环境，如果响应尚未被提交的话

void setStatus(int sc)	设置响应的状态码

HTTP响应头程序示例

接下来的例子使用setIntHeader()方法和setRefreshHeader()方法来模拟一个数字时钟：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Auto Refresh Header Example</title>
</head>
<body>
<center>
<h2>Auto Refresh Header Example</h2>
<%
   // 设置每隔5秒自动刷新
   response.setIntHeader("Refresh", 5);
   // 获取当前时间
   Calendar calendar = new GregorianCalendar();
   String am_pm;
   int hour = calendar.get(Calendar.HOUR);
   int minute = calendar.get(Calendar.MINUTE);
   int second = calendar.get(Calendar.SECOND);
   if(calendar.get(Calendar.AM_PM) == 0)
      am_pm = "AM";
   else
      am_pm = "PM";
   String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
   out.println("Current Time is: " + CT + "\n");
%>
</center>
</body>
</html>
```

将以上代码保存为main.jsp，然后通过浏览器访问它。它将会每隔5秒显示一下系统当前时间。

运行结果如下：

```
Auto Refresh Header Example
Current Time is: 9:44:50 PM
```

您也可以自己动手修改以上代码，试试使用其他的方法，将能得到更深的体会。

JSP HTTP 状态码

HTTP请求与HTTP响应的格式相近，都有着如下结构：

- 以状态行+CRLF（回车换行）开始
- 零行或多行头模块+CRLF
- 一个空行，比如CRLF
- 可选的消息体比如文件，查询数据，查询输出

举例来说，一个服务器响应头看起来就像下面这样：

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

状态行包含HTTP版本，一个状态码，和状态码相对应的短消息。

下表列出了可能会从服务器返回的HTTP状态码和与之关联的消息：

状态码	消息	描述
100	Continue	只有一部分请求被服务器接收，但只要没被服务器拒绝，客户端就会延续这个请求
101	Switching Protocols	服务器交换机协议
200	OK	请求被确认
201	Created	请求时完整的，新的资源被创建
202	Accepted	请求被接受，但未处理完
203	Non-authoritative Information	
204	No Content	
205	Reset Content	
206	Partial Content	

300	Multiple Choices	一个超链接表，用户可以选择一个超链接并访问，最大支持5个超链接
301	Moved Permanently	被请求的页面已经移动到了新的URL下
302	Found	被请求的页面暂时性地移动到了新的URL下
303	See Other	被请求的页面可以在一个不同的URL下找到
304	Not Modified	
305	Use Proxy	
306	*Unused*	已经不再使用此状态码，但状态码被保留
307	Temporary Redirect	被请求的页面暂时性地移动到了新的URL下
400	Bad Request	服务器无法识别请求
401	Unauthorized	被请求的页面需要用户名和密码
402	Payment Required	目前还不能使用此状态码
403	Forbidden	禁止访问所请求的页面
404	Not Found	服务器无法找到所请求的页面
405	Method Not Allowed	请求中所指定的方法不被允许
406	Not Acceptable	服务器只能创建一个客户端无法接受的响应
407	Proxy Authentication Required	在请求被服务前必须认证一个代理服务器
408	Request Timeout	请求时间超过了服务器所能等待的时间，连接被断开
409	Conflict	请求有矛盾的地方
410	Gone	被请求的页面不再可用
411	Length Required	"Content-Length"没有被定义，服务器拒绝接受请求
412	Precondition Failed	请求的前提条件被服务器评估为false
413	Request Entity Too Large	因为请求的实体太大，服务器拒绝接受请求
414	Request-url Too Long	服务器拒绝接受请求，因为URL太长。多出现在把"POST"请求转换为"GET"请求时所附带的大量查询信息
415	Unsupported Media Type	服务器拒绝接受请求，因为媒体类型不被支持

417	Expectation Failed	
500	Internal Server Error	请求不完整，服务器遇见了出乎意料的状况
501	Not Implemented	请求不完整，服务器不提供所需要的功能
502	Bad Gateway	请求不完整，服务器从上游服务器接受了一个无效的响应
503	Service Unavailable	请求不完整，服务器暂时重启或关闭
504	Gateway Timeout	网关超时
505	HTTP Version Not Supported	服务器不支持所指定的HTTP版本

设置HTTP状态码的方法

下表列出了HttpServletResponse 类中用来设置状态码的方法：

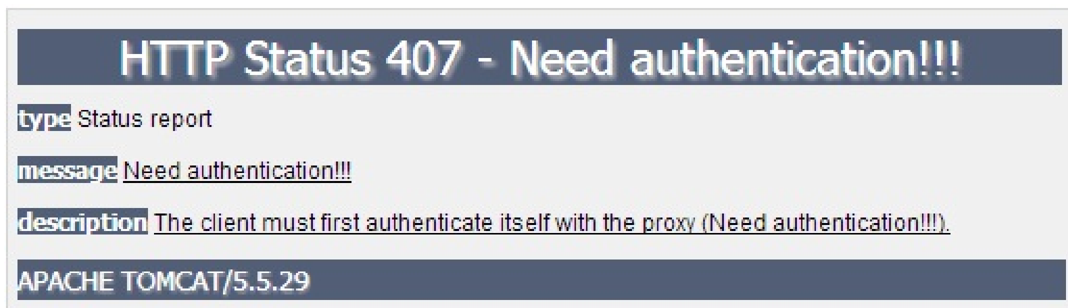
方法	描述
public void setStatus (int statusCode)	此方法可以设置任意的状态码。如果您的响应包含一个特殊的状态码和一个文档，请确保在用PrintWriter返回任何内容前调用setStatus方法
public void sendRedirect(String url)	此方法产生302响应，同时产生一个 *Location* 头告诉URL 一个新的文档
public void sendError(int code, String message)	此方法将一个状态码(通常为 404)和一个短消息，自动插入HTML文档中并发回给客户端

HTTP状态码程序示例

接下来的例子将会发送407错误码给浏览器，然后浏览器将会告诉您"Need authentication!!!"。

```
<html>
<head>
<title>Setting HTTP Status Code</title>
</head>
<body>
<%
   // 设置错误代码，并说明原因
   response.sendError(407, "Need authentication!!!" );
%>
</body>
</html>
```

访问以上JSP页面，将会得到以下结果：



您也可以试试使用其他的状态码，看会不会得到什么意想不到结果。

JSP 表单处理

我们在浏览网页的时候，经常需要向服务器提交信息，并让后台程序处理。浏览器中使用 GET 和 POST 方法向服务器提交数据。

GET 方法

GET方法将请求的编码信息添加在网址后面，网址与编码信息通过"?"号分隔。如下所示：

```
http://www.w3cschool.cc/hello?key1=value1&key2=value2
```

GET方法是浏览器默认传递参数的方法，一些敏感信息，如密码等建议不使用GET方法。

用get时，传输数据的大小有限制 （注意不是参数的个数有限制），最大为1024字节。

POST 方法

一些敏感信息，如密码等我们可以同过POST方法传递，post提交数据是隐式的。

POST提交数据是不可见的，GET是通过在url里面传递的（可以看一下你浏览器的地址栏）。

JSP使用getParameter()来获得传递的参数，getInputStream()方法用来处理客户端的二进制数据流的请求。

JSP 读取表单数据

- **getParameter():** 使用 request.getParameter() 方法来获取表单参数的值。
- **getParameterValues():** 获得如checkbox类（名字相同，但值有多个）的数据。 接收数组变量 ，如checkobx类型
- **getParameterNames():**该方法可以取得所有变量的名称，该方法返回一个Emumeration。
- **getInputStream():**调用此方法来读取来自客户端的二进制数据流。

使用URL的 GET 方法实例

以下是一个简单的URL,并使用GET方法来传递URL中的参数：

```
http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI
```

以下是main.jsp文件的JSP程序用于处理客户端提交的表单数据，我们使用getParameter()方法来获取提交的数据：

```
<html>
<head>
<title>Using GET Method to Read Form Data</title>
</head>
<body>
<center>
<h1>Using GET Method to Read Form Data</h1>
<ul>
<li><p><b>First Name:</b>
   <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
   <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

接下来我们通过浏览器访问*http://localhost:8080/main.jsp?first_name=ZARA&last_name=ALI* 输出结果如下所示：

```
Using GET Method to Read Form Data
First Name: ZARA

Last Name: ALI
```

使用表单的 GET 方法实例

以下是一个简单的HTML表单，该表单通过GET方法将客户端数据提交 到main.jsp文件中：

```
<html>
<body>
<form action="main.jsp" method="GET">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

将以上HTML代码保存到Hello.htm文件中。 将该文件放置于<Tomcat安装目录>/webapps/ROOT 目录下。 通过访问 *http://localhost:8080/Hello.htm*，输出界面如下所示：

First Name:

Last Name: Submit

在"First Name" 与 "Last Name"两个表单中填入信息，并点击"Submit"按钮，它将输出结果。

使用表单的 POST 方法实例

接下来让我们使用POST方法来传递表单数据，修改main.jsp与Hello.htm文件代码，如下所示：

main.jsp文件代码：

```
<html>
<head>
<title>Using GET and POST Method to Read Form Data</title>
</head>
<body>
<center>
<h1>Using GET Method to Read Form Data</h1>
<ul>
<li><p><b>First Name:</b>
   <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
   <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

以下是Hello.htm修改后的代码：

```
<html>
<body>
<form action="main.jsp" method="POST">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

通过浏览器访问 *http://localhost:8080/Hello.htm*，输出如下：

First Name:

Last Name: Submit

在"First Name" 与 "Last Name"两个表单中填入信息，并点击"Submit"按钮，它将输出结果。

传递 Checkbox 数据到JSP程序

复选框 checkbox 可以传递一个甚至多个数据。

以下是一个简单的HTML代码，并将代码保存在CheckBox.htm文件中：

```
<html>
<body>
<form action="main.jsp" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" />
                                                Chemistry
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

以上代码在浏览器访问如下所示：

以下为main.jsp文件代码，用于处理复选框数据：

```
<html>
<head>
<title>Reading Checkbox Data</title>
</head>
<body>
<center>
<h1>Reading Checkbox Data</h1>
<ul>
<li><p><b>Maths Flag:</b>
   <%= request.getParameter("maths")%>
</p></li>
<li><p><b>Physics Flag:</b>
   <%= request.getParameter("physics")%>
</p></li>
<li><p><b>Chemistry Flag:</b>
   <%= request.getParameter("chemistry")%>
</p></li>
</ul>
</body>
</html>
```

以上实例输出结果为：

Maths Physics Chemistry Select Subject

读取所有表单参数

以下我们将使用 HttpServletRequest 的getParameterNames()来读取所有可用的表单参数,该方法可以取得所有变量的名称，该方法返回一个Emumeration。

一旦我们有了一个Enumeration（枚举），我们就可以调用hasMoreElements（）方法来确定何时停止使用和nextElement（）方法来获得每个参数的名称。

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>HTTP Header Request Example</title>
</head>
<body>
<center>
<h2>HTTP Header Request Example</h2>
<table width="100%" border="1" align="center">
<tr bgcolor="#949494">
<th>Param Name</th><th>Param Value(s)</th>
</tr>
<%
   Enumeration paramNames = request.getParameterNames();

   while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.print("<tr><td>" + paramName + "</td>\n");
      String paramValue = request.getHeader(paramName);
      out.println("<td> " + paramValue + "</td></tr>\n");
   }
%>
</table>
</center>
</body>
</html>
```

以下是Hello.htm文件的内容:

```
<html>
<body>
<form action="main.jsp" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" /> Chem
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

现在我们通过浏览器访问 Hello.htm 文件并提交数据，输出结果如下：

Reading All Form Parameters

Param Name	Param Value(s)
maths	on
chemistry	on

你可以尝试使用以上的JSP代码读取其它对象，如文本框，单选按钮或下拉框等等其他形式的数据。

JSP 过滤器

Servlet和JSP中的过滤器都是Java类，它们存在的目的如下：

- 在请求访问后端资源时拦截它
- 管理从服务器返回给客户端的响应

下面列出了多种常用的过滤器类型：

- 认证过滤器
- 数据压缩过滤器
- 加密过滤器
- 触发资源访问事件的过滤器
- 图像转换过滤器
- 登录和验证过滤器
- MIME类型链过滤器
- 令牌过滤器
- 转换XML内容的XSL/T过滤器

过滤器将会被插入进web.xml文件中，然后映射servlet、JSP文件的名字，或URL模式。部署描述文件web.xml可以在 <Tomcat-installation-directory>\conf 目录下找到。

当JSP容器启动网络应用程序时，它会创建每一个过滤器的实例，这些过滤器必须在部署描述文件web.xml中声明，并且按声明的顺序执行。

Servlet过滤器方法

一个过滤器就是一个Java类，它实现了javax.servlet.Filter 接口。javax.servlet.Filter接口定义了三个方法：

方法	描述
public void doFilter (ServletRequest, ServletResponse, FilterChain)	该方法在每次一个请求/响应对因客户端在链的末端请求资源而通过链传递时由容器调用。
public void init(FilterConfig filterConfig)	该方法由 Web 容器调用，指示一个过滤器被放入服务。
public void destroy()	该方法由 Web 容器调用，指示一个过滤器被取出服务。

JSP过滤器示例

这个例子将会打印IP地址和每次访问JSP文件的日期时间。当然，这只是个简单的例子，让您了解一些简单的过滤器用法，但是可以使用这些概念来自行构造更复杂的程序。

```
//  引入Java包
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// 实现 Filter 类
public class LogFilter implements Filter  {
   public void  init(FilterConfig config)
                         throws ServletException{
      // 获取初始化参数
      String testParam = config.getInitParameter("test-param");

      //打印初始化参数
      System.out.println("Test Param: " + testParam);
   }
   public void  doFilter(ServletRequest request,
                 ServletResponse response,
                 FilterChain chain)
                 throws java.io.IOException, ServletException {

      // 获取客户端ip地址
      String ipAddress = request.getRemoteAddr();

      // 输出ip地址及当前时间
      System.out.println("IP "+ ipAddress + ", Time "
                                       + new Date().toString());

      // 传递请求道过滤器链
      chain.doFilter(request,response);
   }
   public void destroy( ){
      /* 在Filter实例在服务器上被移除前调用。*/
   }
}
```

编译LogFilter.java文件，然后将编译后的class文件放在<Tomcat安装目录>/webapps/ROOT/WEB-INF/classes目录下。

web.xml文件中的JSP过滤器映射

过滤器被定义，然后映射成一个URL或JSP文件名，与servlet被定义然后映射的方式差不多。在部署描述文件web.xml中，使用<filter>标签来进行过滤器映射：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

上述过滤器将会应用在所有servlet和JSP程序中，因为我们在配置中指定了" /*"。您也可以指定一个servlet或JSP路径，如果您只想要将过滤器应用在少数几个servlet或JSP程序中的话。

现在，像平常一样访问servlet或JSP页面，您就会发现服务器日志中产生了关于此次访问的记录。您也可以使用Log4J记录器来把日志记录在其它文件中。

使用多重过滤器

您的网络应用程序可以定义很多不同的过滤器。现在，您定义了两个过滤器，AuthenFilter和LogFilter，其它的步骤与前面讲的一样，除非要创建一个不同的映射，就像下面这样：

```
<filter>
   <filter-name>LogFilter</filter-name>
   <filter-class>LogFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter>
   <filter-name>AuthenFilter</filter-name>
   <filter-class>AuthenFilter</filter-class>
   <init-param>
      <param-name>test-param</param-name>
      <param-value>Initialization Paramter</param-value>
   </init-param>
</filter>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

过滤器的应用顺序

在web.xml中<filter>元素的映射顺序决定了容器应用这些过滤器的顺序。要反转应用的顺序，您只需要反转web.xml中<filter>元素的定义顺序就行了。

比如，上面的例子会首先应用 LogFilter然后再应用AuthenFilter，但是下面这个例子将会反转应用的顺序：

```
<filter-mapping>
   <filter-name>AuthenFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>LogFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

JSP Cookies 处理

Cookies是存储在客户机的文本文件，它们保存了大量轨迹信息。在servlet技术基础上，JSP显然能够提供对HTTP cookies的支持。

通常有三个步骤来识别回头客：

- 服务器脚本发送一系列cookies至浏览器。比如名字，年龄，ID号码等等。
- 浏览器在本地机中存储这些信息，以备不时之需。
- 当下一次浏览器发送任何请求至服务器时，它会同时将这些cookies信息发送给服务器，然后服务器使用这些信息来识别用户或者干些其它事情。

本章节将会传授您如何去设置或重设cookie的方法，还有如何访问它们及如何删除它们。

Cookie剖析

Cookies通常在HTTP信息头中设置（虽然JavaScript能够直接在浏览器中设置cookies）。在JSP中，设置一个cookie需要发送如下的信息头给服务器：

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
                 path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```

正如您所见，Set-Cookie信息头包含一个键值对，一个GMT（格林尼治标准）时间，一个路径，一个域名。键值对会被编码为URL。有效期域是个指令，告诉浏览器在什么时候之后就可以清除这个cookie。

如果浏览器被配置成可存储cookies，那么它将会保存这些信息直到过期。如果用户访问的任何页面匹配了cookie中的路径和域名，那么浏览器将会重新将这个cookie发回给服务器。浏览器端的信息头长得就像下面这样：

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

JSP脚本通过request对象中的getCookies()方法来访问这些cookies，这个方法会返回一个Cookie对象的数组。

Servlet Cookies 方法

下表列出了Cookie对象中常用的方法：

方法	描述
public void setDomain(String pattern)	设置cookie的域名，比如w3cschool.cc
public String getDomain()	获取cookie的域名，比如w3cschool.cc
public void setMaxAge(int expiry)	设置cookie有效期，以秒为单位，默认有效期为当前session的存活时间
public int getMaxAge()	获取cookie有效期，以秒为单位，默认为-1，表明cookie会活到浏览器关闭为止
public String getName()	返回 cookie的名称，名称创建后将不能被修改
public void setValue(String newValue)	设置 cookie的值
public String getValue()	获取cookie的值
public void setPath(String uri)	设置cookie 的路径，默认为当前页面目录下的所有URL，还有此目录下的所有子目录
public String getPath()	获取cookie 的路径
public void setSecure(boolean flag)	指明cookie是否要加密传输
public void setComment(String purpose)	设置注释描述 cookie的目的。当浏览器将cookie展现给用户时，注释将会变得非常有用
public String getComment()	返回描述cookie目的的注释，若没有则返回null

使用JSP设置Cookies

使用JSP设置cookie包含三个步骤：

(1)创建一个Cookie对象： 调用Cookie的构造函数，使用一个cookie名称和值做参数，它们都是字符串。

```
Cookie cookie = new Cookie("key","value");
```

请务必牢记，名称和值中都不能包含空格或者如下的字符：

```
[ ] ( ) = , " / ? @ : ;
```

(2) 设置有效期：调用setMaxAge()函数表明cookie在多长时间（以秒为单位）内有效。下面的操作将有效期设为了24小时。

```
cookie.setMaxAge(60*60*24);
```

(3) 将**cookie**发送至**HTTP**响应头中：调用response.addCookie()函数来向HTTP响应头中添加cookies。

```
response.addCookie(cookie);
```

实例演示

```
<%
   // 为 first_name 和 last_name设置cookie
   Cookie firstName = new Cookie("first_name",
               request.getParameter("first_name"));
   Cookie lastName = new Cookie("last_name",
              request.getParameter("last_name"));

   // 设置cookie过期时间为24小时。
   firstName.setMaxAge(60*60*24);
   lastName.setMaxAge(60*60*24);

   // 在响应头部添加cookie
   response.addCookie( firstName );
   response.addCookie( lastName );
%>
<html>
<head>
<title>Setting Cookies</title>
</head>
<body>
<center>
<h1>Setting Cookies</h1>
</center>
<ul>
<li><p><b>First Name:</b>
   <%= request.getParameter("first_name")%>
</p></li>
<li><p><b>Last  Name:</b>
   <%= request.getParameter("last_name")%>
</p></li>
</ul>
</body>
</html>
```

将上面两个文件放在<Tomcat安装目录>/webapps/ROOT目录下，然后访问http://localhost:8080/hello.jsp，将会得到如下输出结果：

First Name: [ ]
Last Name: [ ] Submit

试着输入First Name和Last Name，然后点击提交按钮，它将会在您的屏幕中显示first name和last name，并且设置first name和last name两个cookie，下一次点击提交按钮时会发给服务器。

使用JSP读取Cookies

想要读取cookies，您就需要调用request.getCookies()方法来获得一个javax.servlet.http.Cookie对象的数组，然后遍历这个数组，使用getName()方法和getValue()方法来获取每一个cookie的名称和值。

让我们来读取上个例子中的cookies。

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
   Cookie cookie = null;
   Cookie[] cookies = null;
   // 获取cookies的数据,是一个数组
   cookies = request.getCookies();
   if( cookies != null ){
      out.println("<h2> Found Cookies Name and Value</h2>");
      for (int i = 0; i < cookies.length; i++){
         cookie = cookies[i];
         out.print("Name : " + cookie.getName( ) + ",  ");
         out.print("Value: " + cookie.getValue( )+" <br/>");
      }
  }else{
      out.println("<h2>No cookies founds</h2>");
  }
%>
</body>
</html>
```

如果您把first name cookie设置成"John"，last name设置成"Player"，访问http://localhost:8080/main.jsp，将会得到如下输出结果：

```
Found Cookies Name and Value
Name : first_name, Value: John
Name : last_name, Value: Player
```

使用JSP删除Cookies

删除cookies非常简单。如果您想要删除一个cookie，按照下面给的步骤来做就行了：

- 获取一个已经存在的cookie然后存储在Cookie对象中。
- 将cookie的有效期设置为0。
- 将这个cookie重新添加进响应头中。

实例演示

下面的程序删除一个名为"first_name"的cookie，当您下次运行main.jsp时，first_name将会为null。

```
<html>
<head>
<title>Reading Cookies</title>
</head>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
   Cookie cookie = null;
   Cookie[] cookies = null;
   // 获取当前域名下的cookies，是一个数组
   cookies = request.getCookies();
   if( cookies != null ){
      out.println("<h2> Found Cookies Name and Value</h2>");
      for (int i = 0; i < cookies.length; i++){
         cookie = cookies[i];
         if((cookie.getName( )).compareTo("first_name") == 0 ){
            cookie.setMaxAge(0);
            response.addCookie(cookie);
            out.print("Deleted cookie: " +
            cookie.getName( ) + "<br/>");
         }
         out.print("Name : " + cookie.getName( ) + ",  ");
         out.print("Value: " + cookie.getValue( )+" <br/>");
      }
  }else{
      out.println(
      "<h2>No cookies founds</h2>");
  }
%>
</body>
</html>
```

访问它，将会得到如下输出结果：

```
Cookies Name and Value
Deleted cookie : first_name
Name : first_name, Value: John
Name : last_name, Value: Player
```

再次访问http://localhost:8080/main.jsp，将会得到如下结果：

```
Found Cookies Name and Value
Name : last_name, Value: Player
```

您也可以手动在浏览器中删除cookies。点击Tools菜单项，然后选择Internet Options，点击Delete Cookies，就能删除所有cookies了。

JSP Session

HTTP是无状态协议，这意味着每次客户端检索网页时，都要单独打开一个服务器连接，因此服务器不会记录下先前客户端请求的任何信息。

有三种方法来维持客户端与服务器的会话：

Cookies

网络服务器可以指定一个唯一的session ID作为cookie来代表每个客户端，用来识别这个客户端接下来的请求。

这可能不是一种有效的方式，因为很多时候浏览器并不一定支持cookie，所以我们不建议使用这种方法来维持会话。

隐藏表单域

一个网络服务器可以发送一个隐藏的HTML表单域和一个唯一的session ID，就像下面这样：

```
<input type="hidden" name="sessionid" value="12345">
```

这个条目意味着，当表单被提交时，指定的名称和值将会自动包含在GET或POST数据中。每当浏览器发送一个请求，session_id的值就可以用来保存不同浏览器的轨迹。

这种方式可能是一种有效的方式，但点击<A HREF>标签中的超链接时不会产生表单提交事件，因此隐藏表单域也不支持通用会话跟踪。

重写URL

您可以在每个URL后面添加一些额外的数据来区分会话，服务器能够根据这些数据来关联session标识符。

举例来说，http://w3cschool.cc/file.htm;sessionid=12345， session标识符为sessionid=12345，服务器可以用这个数据来识别客户端。

相比而言，重写URL是更好的方式来，就算浏览器不支持cookies也能工作，但缺点是您必须为每个URL动态指定session ID，就算这是个简单的HTML页面。

session对象

除了以上几种方法外，JSP利用servlet提供的HttpSession接口来识别一个用户，存储这个用户的所有访问信息。

默认情况下，JSP允许会话跟踪，一个新的HttpSession对象将会自动地为新的客户端实例化。禁止会话跟踪需要显式地关掉它，通过将page指令中session属性值设为false来实现，就像下面这样：

```
<%@ page session="false" %>
```

JSP引擎将隐含的session对象暴露给开发者。由于提供了session对象，开发者就可以方便地存储或检索数据。

下表列出了session对象的一些重要方法：

方法	描述
public Object getAttribute(String name)	返回session对象中与指定名称绑定的对象，如果不存在则返回null
public Enumeration getAttributeNames()	返回session对象中所有的对象名称
public long getCreationTime()	返回session对象被创建的时间， 以毫秒为单位，从1970年1月1号凌晨开始算起
public String getId()	返回session对象的ID
public long getLastAccessedTime()	返回客户端最后访问的时间，以毫秒为单位，从1970年1月1号凌晨开始算起
public int getMaxInactiveInterval()	返回最大时间间隔，以秒为单位，servlet 容器将会在这段时间内保持会话打开
public void invalidate()	将session无效化，解绑任何与该session绑定的对象
public boolean isNew(	返回是否为一个新的客户端，或者客户端是否拒绝加入session
public void removeAttribute(String name)	移除session中指定名称的对象
public void setAttribute(String name, Object value)	使用指定的名称和值来产生一个对象并绑定到session中
public void setMaxInactiveInterval(int interval)	用来指定时间，以秒为单位，servlet容器将会在这段时间内保持会话有效

JSP Session应用

这个例子描述了如何使用HttpSession对象来获取创建时间和最后一次访问时间。我们将会为request对象关联一个新的session对象，如果这个对象尚未存在的话。

```
<%@ page import="java.io.*,java.util.*" %>
<%
   // 获取session创建时间
   Date createTime = new Date(session.getCreationTime());
   // 获取最后访问页面的时间
   Date lastAccessTime = new Date(session.getLastAccessedTime());

   String title = "Welcome Back to my website";
   Integer visitCount = new Integer(0);
   String visitCountKey = new String("visitCount");
   String userIDKey = new String("userID");
   String userID = new String("ABCD");

   // 检测网页是否由新的访问用户
   if (session.isNew()){
      title = "Welcome to my website";
      session.setAttribute(userIDKey, userID);
      session.setAttribute(visitCountKey,  visitCount);
   }
   visitCount = (Integer)session.getAttribute(visitCountKey);
   visitCount = visitCount + 1;
   userID = (String)session.getAttribute(userIDKey);
   session.setAttribute(visitCountKey,  visitCount);
%>
<html>
<head>
<title>Session Tracking</title>
</head>
<body>
<center>
<h1>Session Tracking</h1>
</center>
<table border="1" align="center">
<tr bgcolor="#949494">
   <th>Session info</th>
   <th>Value</th>
</tr>
<tr>
   <td>id</td>
   <td><% out.print( session.getId()); %></td>
</tr>
<tr>
   <td>Creation Time</td>
   <td><% out.print(createTime); %></td>
</tr>
<tr>
   <td>Time of Last Access</td>
   <td><% out.print(lastAccessTime); %></td>
</tr>
<tr>
   <td>User ID</td>
   <td><% out.print(userID); %></td>
</tr>
<tr>
   <td>Number of visits</td>
   <td><% out.print(visitCount); %></td>
</tr>
</table>
</body>
</html>
```

试着访问http://localhost:8080/main.jsp，第一次运行时将会得到如下结果：

Welcome to my website

Session Infomation

Session info	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	0

再次访问，将会得到如下结果：

Welcome Back to my website

Session Infomation

info type	value
id	0AE3EC93FF44E3C525B4351B77ABB2D5
Creation Time	Tue Jun 08 17:26:40 GMT+04:00 2010
Time of Last Access	Tue Jun 08 17:26:40 GMT+04:00 2010
User ID	ABCD
Number of visits	1

删除Session数据

当处理完一个用户的会话数据后，您可以有如下选择：

- 移除一个特定的属性：

  调用public void removeAttribute(String name) 方法来移除指定的属性。

- 删除整个会话：

  调用public void invalidate() 方法来使整个session无效。

- 设置会话有效期：

  调用 public void setMaxInactiveInterval(int interval) 方法来设置session超时。

- 登出用户：

  支持servlet2.4版本的服务器，可以调用 logout()方法来登出用户，并且使所有相关的session无效。

- 配置**web.xml**文件：

如果使用的是Tomcat，可以向下面这样配置web.xml文件：

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

超时以分钟为单位，Tomcat中的默认的超时时间是30分钟。

Servlet中的getMaxInactiveInterval( ) 方法以秒为单位返回超时时间。如果在web.xml中配置的是15分钟，则getMaxInactiveInterval( ) 方法将会返回900。

JSP 文件上传

JSP可以通过HTML的form表单上传文件到服务器。 文件类型可以是文本文件、二进制文件、图像文件等其他任何文档。

创建文件上传表单

接下来我们使用HTML标签来创建文件上传表单，以下为要注意的点：

- form表单 **method** 属性必须设置为 **POST** 方法， 不能使用 GET 方法。
- form表单 **enctype** 属性需要设置为 **multipart/form-data**。
- form表单 **action** 属性需要设置为提交到后台处理文件上传的jsp文件地址。例如 **uploadFile.jsp** 程序文件用来处理上传的文件。
- 上传文件元素需要使用 <input .../> 标签，属性设置为 type="file"。如果需要上传多个文件，可以在 <input .../>标签中设置不同的名称。

以下是一个上传文件的表单，实例如下：

```
<html>
<head>
<title>File Uploading Form</title>
</head>
<body>
<h3>File Upload:</h3>
Select a file to upload: <br />
<form action="UploadServlet" method="post"
                        enctype="multipart/form-data">
<input type="file" name="file" size="50" />
<br />
<input type="submit" value="Upload File" />
</form>
</body>
</html>
```

在你本地浏览器访问该文件，显示界面如下所示，在你点击"Upload File"会弹出一个窗口让你选择要上传的文件：

File Upload:
Select a file to upload:

选择文件 未选择文件

Upload File

后台JSP处理脚本

首先我们先定义文件上传后存储在服务上的位置，你可以将路径写在你的程序当中，或者我们可以在web.xml配置文件中通过设置 context-param 元素来设置文件存储的目录，如下所示：

```
<web-app>
....
<context-param>
    <description>Location to store uploaded file</description>
    <param-name>file-upload</param-name>
    <param-value>
         c:\apache-tomcat-5.5.29\webapps\data\
     </param-value>
</context-param>
....
</web-app>
```

以下脚本文件UploadFile.jsp可以处理多个上传的文件，在使用该脚本前，我们需要注意以下几点：

- 以下实例依赖 FileUpload, 所以你需要在你的classpath中引入最新的 **commons-fileupload.x.x.jar** 包文件。 下载地址为：http://commons.apache.org/fileupload/。
- FileUpload 依赖 Commons IO, 所以你需要在你的classpath中引入最新的 **commons-io-x.x.jar** 。 下载地址为： http://commons.apache.org/io/。
- 在测试以下实例时，你需要上传确认上传的文件大小小于 *maxFileSize* 变量设置的大小，否则文件无法上传成功。
- 确保你已经创建了目录 c:\temp 和 c:\apache-tomcat-5.5.29\webapps\data 。

```
<%@ page import="java.io.*,java.util.*, javax.servlet.*" %>
<%@ page import="javax.servlet.http.*" %>
<%@ page import="org.apache.commons.fileupload.*" %>
<%@ page import="org.apache.commons.fileupload.disk.*" %>
<%@ page import="org.apache.commons.fileupload.servlet.*" %>
<%@ page import="org.apache.commons.io.output.*" %>

<%
   File file ;
   int maxFileSize = 5000 * 1024;
   int maxMemSize = 5000 * 1024;
   ServletContext context = pageContext.getServletContext();
   String filePath = context.getInitParameter("file-upload");

   // 验证上传内容了类型
   String contentType = request.getContentType();
   if ((contentType.indexOf("multipart/form-data") >= 0)) {

      DiskFileItemFactory factory = new DiskFileItemFactory();
      // 设置内存中存储文件的最大值
      factory.setSizeThreshold(maxMemSize);
      // 本地存储的数据大于 maxMemSize.
      factory.setRepository(new File("c:\\temp"));

      // 创建一个新的文件上传处理程序
      ServletFileUpload upload = new ServletFileUpload(factory);
      // 设置最大上传的文件大小
      upload.setSizeMax( maxFileSize );
      try{
```

```
            // 解析获取的文件
            List fileItems = upload.parseRequest(request);

            // 处理上传的文件
            Iterator i = fileItems.iterator();

            out.println("<html>");
            out.println("<head>");
            out.println("<title>JSP File upload</title>");
            out.println("</head>");
            out.println("<body>");
            while ( i.hasNext () )
            {
               FileItem fi = (FileItem)i.next();
               if ( !fi.isFormField () )
               {
               // 获取上传文件的参数
               String fieldName = fi.getFieldName();
               String fileName = fi.getName();
               boolean isInMemory = fi.isInMemory();
               long sizeInBytes = fi.getSize();
               // 写入文件
               if( fileName.lastIndexOf("\\") >= 0 ){
               file = new File( filePath ,
               fileName.substring( fileName.lastIndexOf("\\"))) ;
               }else{
               file = new File( filePath ,
               fileName.substring(fileName.lastIndexOf("\\")+1)) ;
               }
               fi.write( file ) ;
               out.println("Uploaded Filename: " + filePath +
               fileName + "<br>");
               }
            }
            out.println("</body>");
            out.println("</html>");
         }catch(Exception ex) {
            System.out.println(ex);
         }
      }else{
         out.println("<html>");
         out.println("<head>");
         out.println("<title>Servlet upload</title>");
         out.println("</head>");
         out.println("<body>");
         out.println("<p>No file uploaded</p>");
         out.println("</body>");
         out.println("</html>");
      }
  %>
```

接下来让我们通过浏览器访问 http://localhost:8080/UploadFile.htm，界面如下所示，并上传文件：



如果你的JSP脚本运行正常，文件将被上传至 c:\apache-tomcat-5.5.29\webapps\data\，你可以打开文件夹看看是否上传成功。

JSP 日期处理

使用JSP最重要的优势之一，就是可以使用所有Java API。本章将会详细地讲述Java中的Date类，它在java.util包下，封装了当前日期和时间。

Date类有两个构造函数。第一个构造函数使用当前日期和时间来初始化对象。

```
Date( )
```

第二个构造函数接受一个参数，这个参数表示从1970年1月1日凌晨至所要表示时间的毫秒数。

```
Date(long millisec)
```

获取Date对象后，您就能够使用下表列出的所有方法：

方法	描述
boolean after(Date date)	如果比给定的日期晚，则返回true，否则返回false
boolean before(Date date)	如果比给定的日期早，则返回true，否则返回false
Object clone( )	获取当前对象的一个副本
int compareTo(Date date)	如果与给定日期相等，则返回0，如果比给定日期早，则返回一个负数，如果比给定日期晚，则返回一个正数
int compareTo(Object obj)	与 compareTo(Date) 方法相同，如果 obj 不是Date类或其子类的对象，抛出ClassCastException异常
boolean equals(Object date)	如果与给定日期相同，则返回true，否则返回false
long getTime( )	返回从1970年1月1日凌晨至此对象所表示时间的毫秒数
int hashCode( )	返回此对象的哈希码
void setTime(long time)	使用给定参数设置时间和日期，参数time表示从1970年1月1日凌晨至time所经过的毫秒数
String toString( )	将此对象转换为字符串并返回这个字符串

获取当前日期和时间

使用JSP编程可以很容易的获取当前日期和时间，只要使用Date对象的toString()方法就行了，就像下面这样：

```
<%@ page import="java.io.*,java.util.*, javax.servlet.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<center>
<h1>Display Current Date & Time</h1>
</center>
<%
   Date date = new Date();
   out.print( "<h2 align=\"center\">" +date.toString()+"</h2>");
%>
</body>
</html>
```

将上面的代码保存在CurrentDate.jsp文件中，然后访问http://localhost:8080/CurrentDate.jsp，运行结果如下：

```
Display Current Date & Time
Mon Jun 21 21:46:49 GMT+04:00 2013
```

刷新http://localhost:8080/CurrentDate.jsp，就可以发现每次刷新所得到的秒数都不相同。

日期比较

就像我在开头所提到的，您可以在JSP脚本中使用任何Java方法。如果您想要比较两个日期，

可以参照下面的方法来做：

- 使用getTime()方法得到毫秒数，然后比较毫秒数就行了。
- 使用before()，after()，equals()方法。比如，new Date(99,2,12).before(new Date(99,2,18))返回true。
- 使用compareTo()方法，这个方法在Comparable接口中定义，在Date中实现。

使用SimpleDateFormat格式化日期

SimpleDateFormat使用一种地区敏感的方式来格式化和解析日期，它允许您使用自定义的模式来格式化日期和时间。

对CurrentDate.jsp稍作修改，得到如下修改后的代码：

```
<%@ page import="java.io.*,java.util.*" %>
<%@ page import="javax.servlet.*,java.text.*" %>
<html>
<head>
<title>Display Current Date & Time</title>
</head>
<body>
<center>
<h1>Display Current Date & Time</h1>
</center>
<%
   Date dNow = new Date( );
   SimpleDateFormat ft =
   new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
   out.print( "<h2 align=\"center\">" + ft.format(dNow) + "</h2>");
%>
</body>
</html>
```

再次编译CurrentDate.jsp，然后访问http://localhost:8080/CurrentDate.jsp，就可以得到如下结果：

```
Display Current Date & Time
Mon 2013.06.21 at 10:06:44 PM GMT+04:00
```

SimpleDateFormat格式码

要指定模式字符串，需要使用下表列出的格式码：

字符	描述	示例
G	时代标识符	AD
y	4位数年份	2001
M	月	July or 07
d	日	10
h	12小时制， A.M./P.M. (1~12)	12
H	24小时制	22
m	分钟	30
s	秒	55
S	毫秒	234
E	星期	Tuesday
D	一年中的某天	360
F	一个月中某星期的某天	2 (second Wed. in July)
w	一年中的某星期	40
W	一个月中的某星期	1
a	A.M./P.M. 标记	PM
k	一天中的某个小时 (1~24)	24
K	一天中的某个小时，A.M./P.M. (0~11)	10
z	时区	Eastern Standard Time
'	文本分隔	Delimiter
"	单引号	`

更多关于Date类的详细信息请查阅Java API文档。

JSP 页面重定向

当需要将文档移动到一个新的位置时，就需要使用JSP重定向了。

最简单的重定向方式就是使用response对象的sendRedirect()方法。这个方法的签名如下：

```
public void response.sendRedirect(String location)
throws IOException
```

这个方法将状态码和新的页面位置作为响应发回给浏览器。您也可以使用setStatus()和setHeader()方法来得到同样的效果：

```
....
String site = "http://www.w3cschool.cc" ;
response.setStatus(response.SC_MOVED_TEMPORARILY);
response.setHeader("Location", site);
....
```

实例演示

这个例子表明了JSP如何进行页面重定向：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Page Redirection</title>
</head>
<body>
<center>
<h1>Page Redirection</h1>
</center>
<%
   // 重定向到新地址
   String site = new String("http://www.w3cschool.cc");
   response.setStatus(response.SC_MOVED_TEMPORARILY);
   response.setHeader("Location", site);
%>
</body>
</html>
```

将以上代码保存在PageRedirecting.jsp文件中，然后访问http://localhost:8080/PageRedirect.jsp，它将会把您带至http://www.w3cschool.cc/。

JSP 点击量统计

有时候我们需要知道某个页面被访问的次数，这时我们就需要在页面上添加页面统计器，页面访问的统计一般在用户第一次载入时累加该页面的访问数上。

要实现一个计数器，您可以利用应用程序隐式对象和相关方法getAttribute()和setAttribute()来实现。

这个对象表示JSP页面的整个生命周期中。当JSP页面初始化时创建此对象，当JSP页面调用jspDestroy()时删除该对象。

以下是在应用中创建变量的语法：

```
application.setAttribute(String Key, Object Value);
```

您可以使用上述方法来设置一个计数器变量及更新该变量的值。读取该变量的方法如下：

```
application.getAttribute(String Key);
```

在页面每次被访问时，你可以读取计数器的当前值，并递增1，然后重新设置，在下一个用户访问时就将新的值显示在页面上。

实例演示

该实例将介绍如何使用JSP来计算特定页面访问的总人数。如果你要计算你网站使用页面的总点击量，那么你就必须将该代码放在所有的JSP页面上。

```
<%@ page import="java.io.*,java.util.*" %>

<html>
<head>
<title>Applcation object in JSP</title>
</head>
<body>
<%
    Integer hitsCount =
      (Integer)application.getAttribute("hitCounter");
    if( hitsCount ==null || hitsCount == 0 ){
       /* 第一次访问 */
       out.println("Welcome to my website!");
       hitsCount = 1;
    }else{
       /* 返回访问值 */
       out.println("Welcome back to my website!");
       hitsCount += 1;
    }
    application.setAttribute("hitCounter", hitsCount);
%>
<center>
<p>Total number of visits: <%= hitsCount%></p>
</center>
</body>
</html>
```

现在我们将上面的代码放置于main.jsp文件上，并访问*http://localhost:8080/main.jsp*文件。你会看到页面会生成个计数器，在我们每次刷新页面时，计数器都会发生变化（每次刷新增加1）。你也可以通过不同的浏览器访问，计数器会在每次访问后增加1。如下所示：

```
Welcome back to my website!

Total number of visits: 12
```

复位计数器

使用以上方法，在web服务器重启后，计数器会被复位为0，即前面保留的数据都会消失，你可以使用一下几种方式解决该问题:

- 在数据库中定义一个用于统计网页访问量的数据表count，字段为hitcount，hitcount默认值为0，将统计数据写入到数据表中。
- 在每次访问时我们读取表中hitcount字段。
- 每次访问时让hitcount自增1。
- 在页面上显示新的 hitcount 值作为页面的访问量。
- 如果你需要统计每个页面的访问量，你可以使用以上逻辑将代码添加到所有页面上。

JSP 自动刷新

想象一下，如果要直播比赛的比分，或股票市场的实时状态，或当前的外汇配给，该怎么实现呢？显然，要实现这种实时功能，您就不得不规律性地刷新页面。

JSP提供了一种机制来使这种工作变得简单，它能够定时地自动刷新页面。

刷新一个页面最简单的方式就是使用response对象的setIntHeader()方法。这个方法的签名如下：

```
public void setIntHeader(String header, int headerValue)
```

这个方法通知浏览器在给定的时间后刷新，时间以秒为单位。

页面自动刷新程序示例

这个例子使用了setIntHeader()方法来设置刷新头，模拟一个数字时钟：

```
<%@ page import="java.io.*,java.util.*" %>
<html>
<head>
<title>Auto Refresh Header Example</title>
</head>
<body>
<center>
<h2>Auto Refresh Header Example</h2>
<%
   // Set refresh, autoload time as 5 seconds
   response.setIntHeader("Refresh", 5);
   // Get current time
   Calendar calendar = new GregorianCalendar();
   String am_pm;
   int hour = calendar.get(Calendar.HOUR);
   int minute = calendar.get(Calendar.MINUTE);
   int second = calendar.get(Calendar.SECOND);
   if(calendar.get(Calendar.AM_PM) == 0)
      am_pm = "AM";
   else
      am_pm = "PM";
   String CT = hour+":"+ minute +":"+ second +" "+ am_pm;
   out.println("Crrent Time: " + CT + "\n");
%>
</center>
</body>
</html>
```

把以上代码保存在main.jsp文件中，访问它。它会每隔5秒钟刷新一次页面并获取系统当前时间。运行结果如下：

```
Auto Refresh Header Example
Current Time is: 9:44:50 PM
```

您也可以自己动手写个更复杂点的程序。

JSP 发送邮件

虽然使用JSP实现邮件发送功能很简单，但是需要有JavaMail API，并且需要安装JavaBean Activation Framework。

- 在这里下载最新版本的 JavaMail。
- 在这里下载最新版本的 JavaBeans Activation Framework(JAF)。

下载并解压这些文件，在根目录下，您将会看到一系列jar包。将mail.jar包和activation.jar包加入CLASSPATH变量中。

发送一封简单的邮件

这个例子展示了如何从您的机器发送一封简单的邮件。它假定localhost已经连接至网络并且有能力发送一封邮件。与此同时，请再一次确认mail.jar包和activation.jar包已经添加进CLASSPATH变量中。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // 收件人的电子邮件
   String to = "abcd@gmail.com";

   // 发件人的电子邮件
   String from = "mcmohd@gmail.com";

   // 假设你是从本地主机发送电子邮件
   String host = "localhost";

   // 获取系统属性对象
   Properties properties = System.getProperties();

   // 设置邮件服务器
   properties.setProperty("mail.smtp.host", host);

   // 获取默认的Session对象。
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // 创建一个默认的MimeMessage对象。
      MimeMessage message = new MimeMessage(mailSession);
      // 设置 From: 头部的header字段
      message.setFrom(new InternetAddress(from));
      // 设置 To: 头部的header字段
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));
      // 设置 Subject: header字段
      message.setSubject("This is the Subject Line!");
      // 现在设置的实际消息
      message.setText("This is actual message");
      // 发送消息
      Transport.send(message);
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>
<html>
<head>
<title>Send Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

现在访问 http://localhost:8080/SendEmail.jsp ， 它将会发送一封邮件给abcd@gmail.com 并显示如下结果：

```
Send Email using JSP
Result: Sent message successfully....
```

如果想要把邮件发送给多人，下面列出的方法可以用来指明多个邮箱地址：

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

参数的描述如下：

- type：这个值将会被设置成TO，CC,或BCC。CC代表副本，BCC代表黑色副本，例子程序中使用的是TO。
- addresses：这是一个邮箱地址的数组，当指定邮箱地址时需要使用InternetAddress()方法。

发送一封HTML邮件

这个例子发送一封简单的HTML邮件。它假定您的localhost已经连接至网络并且有能力发送邮件。与此同时，请再一次确认mail.jar包和activation.jar包已经添加进CLASSPATH变量中。

这个例子和前一个例子非常相似，不过在这个例子中我们使用了setContent()方法，将"text/html"做为第二个参数传给它，用来表明消息中包含了HTML内容。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // 收件人的电子邮件
   String to = "abcd@gmail.com";

   // 发件人的电子邮件
   String from = "mcmohd@gmail.com";

   // 假设你是从本地主机发送电子邮件
   String host = "localhost";

   // 获取系统属性对象
   Properties properties = System.getProperties();

   // 设置邮件服务器
   properties.setProperty("mail.smtp.host", host);

   // 获取默认的Session对象。
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // 创建一个默认的MimeMessage对象。
      MimeMessage message = new MimeMessage(mailSession);
      // 设置 From: 头部的header字段
      message.setFrom(new InternetAddress(from));
      // 设置 To: 头部的header字段
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));
      // 设置 Subject: header字段
      message.setSubject("This is the Subject Line!");

      // 设置 HTML消息
      message.setContent("<h1>This is actual message</h1>",
                            "text/html" );
      // 发送消息
      Transport.send(message);
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
%>
<html>
<head>
<title>Send HTML Email using JSP</title>
</head>
<body>
<center>
<h1>Send Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

现在你可以尝试使用以上JSP文件来发送HTML消息的电子邮件。

在邮件中包含附件

这个例子告诉我们如何发送一封包含附件的邮件。

```
<%@ page import="java.io.*,java.util.*,javax.mail.*"%>
<%@ page import="javax.mail.internet.*,javax.activation.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%
   String result;
   // 收件人的电子邮件
   String to = "abcd@gmail.com";

   // 发件人的电子邮件
   String from = "mcmohd@gmail.com";

   // 假设你是从本地主机发送电子邮件
   String host = "localhost";

   // 获取系统属性对象
   Properties properties = System.getProperties();

   // 设置邮件服务器
   properties.setProperty("mail.smtp.host", host);

   // 获取默认的Session对象。
   Session mailSession = Session.getDefaultInstance(properties);

   try{
      // 创建一个默认的MimeMessage对象。
      MimeMessage message = new MimeMessage(mailSession);

      // 设置 From: 头部的header字段
      message.setFrom(new InternetAddress(from));

      // 设置 To: 头部的header字段
      message.addRecipient(Message.RecipientType.TO,
                               new InternetAddress(to));

      // 设置 Subject: header字段
      message.setSubject("This is the Subject Line!");

      // 创建消息部分
      BodyPart messageBodyPart = new MimeBodyPart();

      // 填充消息
      messageBodyPart.setText("This is message body");

      // 创建多媒体消息
      Multipart multipart = new MimeMultipart();

      // 设置文本消息部分
      multipart.addBodyPart(messageBodyPart);

      // 附件部分
      messageBodyPart = new MimeBodyPart();
      String filename = "file.txt";
      DataSource source = new FileDataSource(filename);
      messageBodyPart.setDataHandler(new DataHandler(source));
      messageBodyPart.setFileName(filename);
      multipart.addBodyPart(messageBodyPart);

      // 发送完整消息
      message.setContent(multipart );

      // 发送消息
      Transport.send(message);
      String title = "Send Email";
      result = "Sent message successfully....";
   }catch (MessagingException mex) {
      mex.printStackTrace();
      result = "Error: unable to send message....";
   }
```

```
%>
<html>
<head>
<title>Send Attachement Email using JSP</title>
</head>
<body>
<center>
<h1>Send Attachement Email using JSP</h1>
</center>
<p align="center">
<%
   out.println("Result: " + result + "\n");
%>
</p>
</body>
</html>
```

用户认证部分

如果邮件服务器需要用户名和密码来进行用户认证的话，可以像下面这样来设置：

```
 props.setProperty("mail.user", "myuser");
 props.setProperty("mail.password", "mypwd");
```

使用表单发送邮件

使用HTML表单接收一封邮件，并通过request对象获取所有邮件信息：

```
String to = request.getParameter("to");
String from = request.getParameter("from");
String subject = request.getParameter("subject");
String messageText = request.getParameter("body");
```

获取以上信息后，您就可以使用前面提到的例子来发送邮件了。

JSP 高级教程

JSP 标准标签库（JSTL）

JSP标准标签库（JSTL）是一个JSP标签集合，它封装了JSP应用的通用核心功能。

JSTL支持通用的、结构化的任务，比如迭代，条件判断，XML文档操作，国际化标签，SQL标签。 除了这些，它还提供了一个框架来使用集成JSTL的自定义标签。

根据JSTL标签所提供的功能，可以将其分为5个类别。

- 核心标签
- 格式化标签
- **SQL** 标签
- **XML** 标签
- **JSTL** 函数

JSTL 库安装

Apache Tomcat安装JSTL 库步骤如下：

- 从Apache的标准标签库中下载的二进包(jakarta-taglibs-standard-current.zip)。下载地址：http://archive.apache.org/dist/jakarta/taglibs/standard/binaries/
- 下载jakarta-taglibs-standard-1.1.1.zip 包并解压，将jakarta-taglibs-standard-1.1.1/lib/下的两个jar文件：standard.jar和jstl.jar文件拷贝到/WEB-INF/lib/下。

使用任何库，你必须在每个JSP文件中的头部包含<taglib>标签。

核心标签

核心标签是最常用的JSTL标签。引用核心标签库的语法如下：

```
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
```

标签	描述
<c:out>	用于在JSP中显示数据，就像<%= ... >
<c:set>	用于保存数据
<c:remove>	用于删除数据
<c:catch>	用来处理产生错误的异常状况，并且将错误信息储存起来
<c:if>	与我们在一般程序中用的if一样
<c:choose>	本身只当做<c:when>和<c:otherwise>的父标签
<c:when>	<c:choose>的子标签，用来判断条件是否成立
<c:otherwise>	<c:choose>的子标签，接在<c:when>标签后，当<c:when>标签判断为false时被执行
<c:import>	检索一个绝对或相对 URL，然后将其内容暴露给页面
<c:forEach>	基础迭代标签，接受多种集合类型
<c:forTokens>	根据指定的分隔符来分隔内容并迭代输出
<c:param>	用来给包含或重定向的页面传递参数
<c:redirect>	重定向至一个新的URL.
<c:url>	使用可选的查询参数来创造一个URL

格式化标签

JSTL格式化标签用来格式化并输出文本、日期、时间、数字。引用格式化标签库的语法如下：

```
<%@ taglib prefix="fmt"
           uri="http://java.sun.com/jsp/jstl/fmt" %>
```

标签	描述
<fmt:formatNumber>	使用指定的格式或精度格式化数字
<fmt:parseNumber>	解析一个代表着数字，货币或百分比的字符串
<fmt:formatDate>	使用指定的风格或模式格式化日期和时间
<fmt:parseDate>	解析一个代表着日期或时间的字符串
<fmt:bundle>	绑定资源
<fmt:setLocale>	指定地区
<fmt:setBundle>	绑定资源
<fmt:timeZone>	指定时区
<fmt:setTimeZone>	指定时区
<fmt:message>	显示资源配置文件信息
<fmt:requestEncoding>	设置request的字符编码

SQL标签

JSTL SQL标签库提供了与关系型数据库（Oracle，MySQL，SQL Server等等）进行交互的标签。引用SQL标签库的语法如下：

```
<%@ taglib prefix="sql"
           uri="http://java.sun.com/jsp/jstl/sql" %>
```

标签	描述
<sql:setDataSource>	指定数据源
<sql:query>	运行SQL查询语句
<sql:update>	运行SQL更新语句
<sql:param>	将SQL语句中的参数设为指定值
<sql:dateParam>	将SQL语句中的日期参数设为指定的java.util.Date 对象值
<sql:transaction>	在共享数据库连接中提供嵌套的数据库行为元素，将所有语句以一个事务的形式来运行

XML 标签

JSTL XML标签库提供了创建和操作XML文档的标签。引用XML标签库的语法如下：

```
<%@ taglib prefix="x"
           uri="http://java.sun.com/jsp/jstl/xml" %>
```

在使用xml标签前，你必须将XML 和 XPath 的相关包拷贝至你的<Tomcat 安装目录>\lib下:

XercesImpl.jar:

下载地址： http://www.apache.org/dist/xerces/j/

xalan.jar:

下载地址： http://xml.apache.org/xalan-j/index.html

标签	描述
<x:out>	与<%= ... >,类似，不过只用于XPath表达式
<x:parse>	解析 XML 数据
<x:set>	设置XPath表达式
<x:if>	判断XPath表达式，若为真，则执行本体中的内容，否则跳过本体
<x:forEach>	迭代XML文档中的节点
<x:choose>	<x:when>和<x:otherwise>的父标签
<x:when>	<x:choose>的子标签，用来进行条件判断
<x:otherwise>	<x:choose>的子标签，当<x:when>判断为false时被执行
<x:transform>	将XSL转换应用在XML文档中
<x:param>	与<x:transform>共同使用，用于设置XSL样式表

JSTL函数

JSTL包含一系列标准函数，大部分是通用的字符串处理函数。引用JSTL函数库的语法如下：

```
<%@ taglib prefix="fn"
           uri="http://java.sun.com/jsp/jstl/functions" %>
```

函数	描述
fn:contains()	测试输入的字符串是否包含指定的子串
fn:containsIgnoreCase()	测试输入的字符串是否包含指定的子串，大小写不敏感
fn:endsWith()	测试输入的字符串是否以指定的后缀结尾
fn:escapeXml()	跳过可以作为XML标记的字符
fn:indexOf()	返回指定字符串在输入字符串中出现的位置
fn:join()	将数组中的元素合成一个字符串然后输出
fn:length()	返回字符串长度
fn:replace()	将输入字符串中指定的位置替换为指定的字符串然后返回
fn:split()	将字符串用指定的分隔符分隔然后组成一个子字符串数组并返回
fn:startsWith()	测试输入字符串是否以指定的前缀开始
fn:substring()	返回字符串的子集
fn:substringAfter()	返回字符串在指定子串之后的子集
fn:substringBefore()	返回字符串在指定子串之前的子集
fn:toLowerCase()	将字符串中的字符转为小写
fn:toUpperCase()	将字符串中的字符转为大写
fn:trim()	移除首位的空白符

JSP 连接数据库

本章节假设您已经对JDBC有一定的了解。在开始学习JSP数据库访问前，请确保JDBC环境已经正确配置。

首先，让我们按照下面的步骤来创建一个简单的表并插入几条简单的记录：

创建表

在数据库中创建一个Employees表，步骤如下：

步骤1：

打开CMD，然后进入数据库安装目录：

```
C:\>
C:\>cd Program Files\MySQL\bin
C:\Program Files\MySQL\bin>
```

步骤2：

```
C:\Program Files\MySQL\bin>mysql -u root -p
Enter password: ********
mysql>
```

步骤3：

在TEST数据库中创建Employee表：

```
mysql> use TEST;
mysql> create table Employees
    (
     id int not null,
     age int not null,
     first varchar (255),
     last varchar (255)
    );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

插入数据记录

创建好Employee表后，往表中插入几条记录：

```
mysql> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

SELECT操作

接下来的这个例子告诉我们如何使用JSTL SQL标签来运行SQL SELECT语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>SELECT Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28

INSERT操作

这个例子告诉我们如何使用JSTL SQL标签来运行SQL INSERT语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>JINSERT Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<sql:update dataSource="${snapshot}" var="result">
INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');
</sql:update>

<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30
103	Sumit	Mittal	28
104	Nuha	Ali	2

DELETE操作

这个例子告诉我们如何使用JSTL SQL标签来运行SQL DELETE语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>DELETE Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<c:set var="empId" value="103"/>

<sql:update dataSource="${snapshot}" var="count">
  DELETE FROM Employees WHERE Id = ?
  <sql:param value="${empId}" />
</sql:update>

<sql:query dataSource="${snapshot}" var="result">
   SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Khan	30

UPDATE操作

这个例子告诉我们如何使用JSTL SQL标签来运行SQL UPDATE语句：

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>
<%@ page import="javax.servlet.http.*,javax.servlet.*" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>
<head>
<title>DELETE Operation</title>
</head>
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://localhost/TEST"
     user="root"  password="pass123"/>

<c:set var="empId" value="102"/>

<sql:update dataSource="${snapshot}" var="count">
  UPDATE Employees SET last = 'Ali'
  <sql:param value="${empId}" />
</sql:update>

<sql:query dataSource="${snapshot}" var="result">
   SELECT * from Employees;
</sql:query>

<table border="1" width="100%">
<tr>
   <th>Emp ID</th>
   <th>First Name</th>
   <th>Last Name</th>
   <th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
   <td><c:out value="${row.id}"/></td>
   <td><c:out value="${row.first}"/></td>
   <td><c:out value="${row.last}"/></td>
   <td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>

</body>
</html>
```

访问这个JSP例子，运行结果如下：

Emp ID	First Name	Last Name	Age
100	Zara	Ali	18
101	Mahnaz	Fatma	25
102	Zaid	Ali	30

JSP XML 数据处理

当通过HTTP发送XML数据时，就有必要使用JSP来处理传入和流出的XML文档了，比如RSS文档。作为一个XML文档，它仅仅只是一堆文本而已，使用JSP创建XML文档并不比创建一个HTML文档难。

使用JSP发送XML

使用JSP发送XML内容就和发送HTML内容一样。唯一的不同就是您需要把页面的context属性设置为text/xml。要设置context属性，使用<%@page % >命令，就像这样：

```
<%@ page contentType="text/xml" %>
```

接下来这个例子向浏览器发送XML内容：

```
<%@ page contentType="text/xml" %>

<books>
   <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
   </book>
</books>
```

使用不同的浏览器来访问这个例子，看看这个例子所呈现的文档树。

在JSP中处理XML

在使用JSP处理XML之前，您需要将与XML 和XPath相关的两个库文件放在<Tomcat Installation Directory>\lib目录下：

- XercesImpl.jar：在这下载http://www.apache.org/dist/xerces/j/
- xalan.jar：在这下载http://xml.apache.org/xalan-j/index.html

books.xml文件:

```
<books>
<book>
  <name>Padam History</name>
  <author>ZARA</author>
  <price>100</price>
</book>
<book>
  <name>Great Mistry</name>
  <author>NUHA</author>
  <price>2000</price>
</book>
</books>
```

main.jsp文件：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<html>
<head>
  <title>JSTL x:parse Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:import var="bookInfo" url="http://localhost:8080/books.xml"/>

<x:parse xml="${bookInfo}" var="output"/>
<b>The title of the first book is</b>:
<x:out select="$output/books/book[1]/name" />
<br>
<b>The price of the second book</b>:
<x:out select="$output/books/book[2]/price" />

</body>
</html>
```

访问http://localhost:8080/main.jsp，运行结果如下：

```
BOOKS INFO:
The title of the first book is:Padam History
The price of the second book: 2000
```

使用JSP格式化XML

这个是XSLT样式表style.xsl文件：

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
  <html>
  <body>
   <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="books">
  <table border="1" width="100%">
    <xsl:for-each select="book">
      <tr>
        <td>
          <i><xsl:value-of select="name"/></i>
        </td>
        <td>
          <xsl:value-of select="author"/>
        </td>
        <td>
          <xsl:value-of select="price"/>
        </td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

这个是main.jsp文件：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<html>
<head>
  <title>JSTL x:transform Tags</title>
</head>
<body>
<h3>Books Info:</h3>
<c:set var="xmltext">
  <books>
    <book>
      <name>Padam History</name>
      <author>ZARA</author>
      <price>100</price>
    </book>
    <book>
      <name>Great Mistry</name>
      <author>NUHA</author>
      <price>2000</price>
    </book>
  </books>
</c:set>

<c:import url="http://localhost:8080/style.xsl" var="xslt"/>
<x:transform xml="${xmltext}" xslt="${xslt}"/>

</body>
</html>
```

运行结果如下：

BOOKS INFO:

Padam History	ZARA	100
Great Mistry	NUHA	2000

更多关于使用JSTL处理XML的内容请查阅JSP标准标签库。

JSP JavaBean

JavaBean是特殊的Java类，使用J ava语言书写，并且遵守JavaBeans API规范。

接下来给出的是JavaBean与其它Java类相比而言独一无二的特征：

- 提供一个默认的无参构造函数。
- 需要被序列化并且实现了Serializable接口。
- 可能有一系列可读写属性。
- 可能有一系列的"getter"或"setter"方法。

JavaBeans属性

一个JavaBean对象的属性应该是可访问的。这个属性可以是任意合法的Java数据类型，包括自定义Java类。

一个JavaBean对象的属性可以是可读写，或只读，或只写。JavaBean对象的属性通过JavaBean实现类中提供的两个方法来访问：

方法	描述
get**PropertyName**()	举例来说，如果属性的名称为myName，那么这个方法的名字就要写成getMyName()来读取这个属性。这个方法也称为访问器。
set**PropertyName**()	举例来说，如果属性的名称为myName，那么这个方法的名字就要写成setMyName()来写入这个属性。这个方法也称为写入器。

一个只读的属性只提供getPropertyName()方法，一个只写的属性只提供setPropertyName()方法。

JavaBeans程序示例

这是StudentBean.java文件：

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable
{
   private String firstName = null;
   private String lastName = null;
   private int age = 0;

   public StudentsBean() {
   }
   public String getFirstName(){
      return firstName;
   }
   public String getLastName(){
      return lastName;
   }
   public int getAge(){
      return age;
   }
   public void setFirstName(String firstName){
      this.firstName = firstName;
   }
   public void setLastName(String lastName){
      this.lastName = lastName;
   }
   public void setAge(Integer age){
      this.age = age;
   }
}
```

编译StudentBean.java文件，在本章最后的例子中将会使用到它。

访问JavaBeans

<jsp:useBean>标签可以在JSP中声明一个JavaBean，然后使用。声明后，JavaBean对象就成了脚本变量，可以通过脚本元素或其他自定义标签来访问。<jsp:useBean>标签的语法格式如下：

```
<jsp:useBean id="bean's name" scope="bean's scope" typeSpec/>
```

其中，根据具体情况，scope的值可以是page，request，session或application。id值可任意只要不和同一JSP文件中其它<jsp:useBean>中id值一样就行了。

接下来给出的是<jsp:useBean>标签的一个简单的用法：

```
<html>
<head>
<title>useBean Example</title>
</head>
<body>

<jsp:useBean id="date" class="java.util.Date" />
<p>The date/time is <%= date %>

</body>
</html>
```

它将会产生如下结果：

```
The date/time is Thu Sep 30 11:18:11 GST 2013
```

访问JavaBeans对象的属性

在<jsp:useBean>标签主体中使用<jsp:getProperty/>标签来调用getter方法，使用<jsp:setProperty/>标签来调用setter方法，语法格式如下：

```
<jsp:useBean id="id" class="bean's class" scope="bean's scope">
   <jsp:setProperty name="bean's id" property="property name"
                    value="value"/>
   <jsp:getProperty name="bean's id" property="property name"/>
   ...........
</jsp:useBean>
```

name属性指的是Bean的id属性。property属性指的是想要调用的getter或setter方法。

接下来给出使用以上语法进行属性访问的一个简单例子：

```
<html>
<head>
<title>get and set properties Example</title>
</head>
<body>

<jsp:useBean id="students"
                    class="com.tutorialspoint.StudentsBean">
   <jsp:setProperty name="students" property="firstName"
                    value="Zara"/>
   <jsp:setProperty name="students" property="lastName"
                    value="Ali"/>
   <jsp:setProperty name="students" property="age"
                    value="10"/>
</jsp:useBean>

<p>Student First Name:
   <jsp:getProperty name="students" property="firstName"/>
</p>
<p>Student Last Name:
   <jsp:getProperty name="students" property="lastName"/>
</p>
<p>Student Age:
   <jsp:getProperty name="students" property="age"/>
</p>

</body>
</html>
```

将StudentBean.class加入CLASSPATH环境变量中，然后访问以上JSP，运行结果如下：

```
Student First Name: Zara

Student Last Name: Ali

Student Age: 10
```

JSP 自定义标签

自定义标签是用户定义的JSP语言元素。当JSP页面包含一个自定义标签时将被转化为servlet，标签转化为对被 称为tag handler的对象的操作，即当servlet执行时Web container调用那些操作。

JSP标签扩展可以让你创建新的标签并且可以直接插入到一个JSP页面。 JSP 2.0规范中引入Simple Tag Handlers来编写这些自定义标记。

你可以继承SimpleTagSupport类并重写的doTag()方法来开发一个最简单的自定义标签。

创建"Hello"标签

接下来，我们想创建一个自定义标签叫作<ex:Hello>，标签格式为：

```
<ex:Hello />
```

要创建自定义的JSP标签，你首先必须创建处理标签的Java类。所以，让我们创建一个HelloTag类，如下所示：

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.println("Hello Custom Tag!");
  }
}
```

以下代码重写了doTag()方法，方法中使用了getJspContext()方法来获取当前的JspContext对象，并将"Hello Custom Tag!"传递给JspWriter对象。

编译以上类，并将其复制到环境变量CLASSPATH目录中。最后创建如下标签库：<Tomcat安装目录>webapps\ROOT\WEB-INF\custom.tld。

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

接下来，我们就可以在JSP文件中使用Hello标签：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello/>
  </body>
</html>
```

以上程序输出结果为：

```
Hello Custom Tag!
```

访问标签体

你可以像标准标签库一样在标签中包含消息内容。如我们要在我们自定义的Hello中包含内容，格式如下：

```
<ex:Hello>
   This is message body
</ex:Hello>
```

我们可以修改标签处理类文件，代码如下：

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

   StringWriter sw = new StringWriter();
   public void doTag()
      throws JspException, IOException
    {
       getJspBody().invoke(sw);
       getJspContext().getOut().println(sw.toString());
    }

}
```

接下来我们需要修改TLD文件，如下所示：

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

现在我们可以在JSP使用修改后的标签，如下所示:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello>
        This is message body
    </ex:Hello>
  </body>
</html>
```

以上程序输出结果如下所示：

```
This is message body
```

自定义标签属性

你可以在自定义标准中设置各种属性，要接收属性，值自定义标签类必须实现setter方法，JavaBean 中的setter方法如下所示：

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

   private String message;

   public void setMessage(String msg) {
      this.message = msg;
   }

   StringWriter sw = new StringWriter();

   public void doTag()
      throws JspException, IOException
    {
       if (message != null) {
          /* 从属性中使用消息 */
          JspWriter out = getJspContext().getOut();
          out.println( message );
       }
       else {
          /* 从内容体中使用消息 */
          getJspBody().invoke(sw);
          getJspContext().getOut().println(sw.toString());
       }
   }

}
```

属性的名称是"message"，所以setter方法??是的setMessage()。现在让我们在TLD文件中使用的<attribute>元素添加此属性：

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>2.0</jsp-version>
  <short-name>Example TLD with Body</short-name>
  <tag>
    <name>Hello</name>
    <tag-class>com.tutorialspoint.HelloTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
       <name>message</name>
    </attribute>
  </tag>
</taglib>
```

现在我们就可以在JSP文件中使用message属性了，如下所示：

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
  <head>
    <title>A sample custom tag</title>
  </head>
  <body>
    <ex:Hello message="This is custom tag" />
  </body>
</html>
```

以上实例数据输出结果为：

```
This is custom tag
```

你还可以包含以下属性：

属性	描述
name	定义属性的名称。每个标签的是属性名称必须是唯一的。
required	指定属性是否是必须的或者可选的,如果设置为false为可选。
rtexprvalue	声明在运行表达式时，标签属性是否有效。
type	定义该属性的Java类类型 。默认指定为 **String**
description	描述信息
fragment	如果声明了该属性,属性值将被视为一个 **JspFragment**。

以下是指定相关的属性实例：

```
.....
    <attribute>
      <name>attribute_name</name>
      <required>false</required>
      <type>java.util.Date</type>
      <fragment>false</fragment>
    </attribute>
.....
```

如果你使用了两个属性，修改TLD文件，如下所示：

```
.....
    <attribute>
      <name>attribute_name1</name>
      <required>false</required>
      <type>java.util.Boolean</type>
      <fragment>false</fragment>
    </attribute>
    <attribute>
      <name>attribute_name2</name>
      <required>true</required>
      <type>java.util.Date</type>
    </attribute>
.....
```

JSP 表达式语言

JSP表达式语言（EL）使得访问存储在JavaBean中的数据变得非常简单。JSP EL既可以用来创建算术表达式也可以用来创建逻辑表达式。在JSP EL表达式内可以使用整型数，浮点数，字符串，常量true、false，还有null。

一个简单的语法

典型的，当您需要在JSP标签中指定一个属性值时，只需要简单地使用字符串即可：

```
<jsp:setProperty name="box" property="perimeter" value="100"/>
```

JSP EL允许您指定一个表达式来表示属性值。一个简单的表达式语法如下：

```
${expr}
```

其中，expr指的是表达式。在JSP EL中通用的操作符是"."和"[]"。这两个操作符允许您通过内嵌的JSP对象访问各种各样的JavaBean属性。

举例来说，上面的<jsp:setProperty>标签可以使用表达式语言改写成如下形式：

```
<jsp:setProperty name="box" property="perimeter"
                 value="${2*box.width+2*box.height}"/>
```

当JSP编译器在属性中见到"${}"格式后，它会产生代码来计算这个表达式，并且产生一个替代品来代替表达式的值。

您也可以在标签的模板文本中使用表达式语言。比如<jsp:text>标签简单地将其主体中的文本插入到JSP输出中：

```
<jsp:text>
<h1>Hello JSP!</h1>
</jsp:text>
```

现在，在<jsp:text>标签主体中使用表达式，就像这样：

```
<jsp:text>
Box Perimeter is: ${2*box.width + 2*box.height}
</jsp:text>
```

在EL表达式中可以使用圆括号来组织子表达式。比如${(1 + 2) 3}等于9，但是${1 + (2 3)} 等于7。

想要停用对EL表达式的评估的话，需要使用page指令将isELIgnored属性值设为true：

```
<%@ page isELIgnored ="true|false" %>
```

这样，EL表达式就会被忽略。若设为false，则容器将会计算EL表达式。

EL中的基础操作符

EL表达式支持大部分Java所提供的算术和逻辑操作符：

操作符	描述
.	访问一个Bean属性或者一个映射条目
[]	访问一个数组或者链表的元素
( )	组织一个子表达式以改变优先级
+	加
-	减或负
*	乘
/ or div	除
% or mod	取模
== or eq	测试是否相等
!= or ne	测试是否不等
< or lt	测试是否小于
> or gt	测试是否大于
<= or le	测试是否小于等于
>= or gt	测试是否大于等于
&& or and	测试逻辑与
|| or or	测试逻辑或
! or not	测试取反
empty	测试是否空值

JSP EL中的函数

JSP EL允许您在表达式中使用函数。这些函数必须被定义在自定义标签库中。函数的使用语法如下：

```
${ns:func(param1, param2, ...)}
```

ns指的是命名空间（namespace），func指的是函数的名称，param1指的是第一个参数，param2指的是第二个参数，以此类推。比如，有函数fn:length，在JSTL库中定义，可以像下面这样来获取一个字符串的长度：

```
${fn:length("Get my length")}
```

要使用任何标签库中的函数，您需要将这些库安装在服务器中，然后使用<taglib>标签在JSP文件中包含这些库。

JSP EL隐含对象

JSP EL支持下表列出的隐含对象：

隐含对象	描述
pageScope	page 作用域
requestScope	request 作用域
sessionScope	session 作用域
applicationScope	application 作用域
param	Request 对象的参数，字符串
paramValues	Request对象的参数，字符串集合
header	HTTP 信息头，字符串
headerValues	HTTP 信息头，字符串集合
initParam	上下文初始化参数
cookie	Cookie值
pageContext	当前页面的pageContext

您可以在表达式中使用这些对象，就像使用变量一样。接下来会给出几个例子来更好的理解这个概念。

pageContext对象

pageContext对象是JSP中pageContext对象的引用。通过pageContext对象，您可以访问request对象。比如，访问request对象传入的查询字符串，就像这样：

```
${pageContext.request.queryString}
```

Scope对象

pageScope，requestScope，sessionScope，applicationScope变量用来访问存储在各个作用域层次的变量。

举例来说，如果您需要显式访问在applicationScope层的box变量，可以这样来访问：applicationScope.box。

param和paramValues对象

param和paramValues对象用来访问参数值，通过使用request.getParameter方法和request.getParameterValues方法。

举例来说，访问一个名为order的参数，可以这样使用表达式：${param.order}，或者${param["order"]}。

接下来的例子表明了如何访问request中的username参数：

```
<%@ page import="java.io.*,java.util.*" %>
<%
    String title = "Accessing Request Param";
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>${param["username"]}</p>
</div>
</body>
</html>
```

param对象返回单一的字符串，而paramValues对象则返回一个字符串数组。

header和headerValues对象

header和headerValues对象用来访问信息头，通过使用 request.getHeader方法和request.getHeaders方法。

举例来说，要访问一个名为user-agent的信息头，可以这样使用表达式：${header.user-agent}，或者${header["user-agent"]}。

接下来的例子表明了如何访问user-agent信息头：

```
<%@ page import="java.io.*,java.util.*" %>
<%
    String title = "User Agent Example";
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>${header["user-agent"]}</p>
</div>
</body>
</html>
```

运行结果如下：

User Agent Example

Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; HPNTDF; .NET4.0C; InfoPath.2)

header对象返回单一值，而headerValues则返回一个字符串数组。

JSP 异常处理

当编写JSP程序的时候，程序员可能会遗漏一些BUG，这些BUG可能会出现在程序的任何地方。JSP代码中通常有以下几类异常:

- 检查型异常:检查型异常就是一个典型的用户错误或者一个程序员无法预见的错误。举例来说，如果一个文件将要被打开，但是无法找到这个文件，则一个异常被抛出。这些异常不能再编译期被简单地忽略。
- 运行时异常:一个运行时异常可能已经被程序员避免，这种异常在编译期将会被忽略。
- 错误:这里没有异常，但问题是它超出了用户或者程序员的控制范围。错误通常会在代码中被忽略，您几乎不能拿它怎么样。举例来或，栈溢出错误。这些错误都会在编译期被忽略。

本节将会给出几个简单而优雅的方式来处理运行时异常和错误。

使用Exception对象

exception对象是Throwable子类的一个实例，只在错误页面中可用。下表列出了Throwable类中一些重要的方法:

方法	描述
public String getMessage()	返回异常的信息。这个信息在Throwable构造函数中被初始化
public ThrowablegetCause()	返回引起异常的原因，类型为Throwable对象
public String toString()	返回类名
public void printStackTrace()	将异常栈轨迹输出至System.err
public StackTraceElement [] getStackTrace()	以栈轨迹元素数组的形式返回异常栈轨迹
public ThrowablefillInStackTrace()	使用当前栈轨迹填充Throwable对象

JSP提供了可选项来为每个JSP页面指定错误页面。无论何时页面抛出了异常，JSP容器都会自动地调用错误页面。

接下来的例子为main.jsp指定了一个错误页面。使用<%@page errorPage="XXXXX"%>指令指定一个错误页面。

```
<%@ page errorPage="ShowError.jsp" %>

<html>
<head>
   <title>Error Handling Example</title>
</head>
<body>
<%
   // Throw an exception to invoke the error page
   int x = 1;
   if (x == 1)
   {
      throw new RuntimeException("Error condition!!!");
   }
%>
</body>
</html>
```

现在，编写ShowError.jsp文件如下:

```
<%@ page isErrorPage="true" %>
<html>
<head>
<title>Show Error Page</title>
</head>
<body>
<h1>Opps...</h1>
<p>Sorry, an error occurred.</p>
<p>Here is the exception stack trace: </p>
<pre>
<% exception.printStackTrace(response.getWriter()); %>
```

注意到，ShowError.jsp文件使用了<%@page isErrorPage="true"%>指令，这个指令告诉JSP编译器需要产生一个异常实例变量。

现在试着访问main.jsp页面，它将会产生如下结果:

```
java.lang.RuntimeException: Error condition!!!
......

Opps...
Sorry, an error occurred.

Here is the exception stack trace:
```

在错误页面中使用JSTL标签

可以利用JSTL标签来编写错误页面ShowError.jsp。这个例子中的代码与上例代码的逻辑几乎一样，但是本例的代码有更好的结构，并且能够提供更多信息:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page isErrorPage="true" %>
<html>
<head>
<title>Show Error Page</title>
</head>
<body>
<h1>Opps...</h1>
<table width="100%" border="1">
<tr valign="top">
<td width="40%"><b>Error:</b></td>
<td>${pageContext.exception}</td>
</tr>
<tr valign="top">
<td><b>URI:</b></td>
<td>${pageContext.errorData.requestURI}</td>
</tr>
<tr valign="top">
<td><b>Status code:</b></td>
<td>${pageContext.errorData.statusCode}</td>
</tr>
<tr valign="top">
<td><b>Stack trace:</b></td>
<td>
<c:forEach var="trace"
         items="${pageContext.exception.stackTrace}">
<p>${trace}</p>
</c:forEach>
</td>
</tr>
</table>
</body>
</html>
```

运行结果如下:

Opps...

Error:	java.lang.RuntimeException: Error condition!!!
URI:	/main.jsp
Status code:	500
Stack trace:	org.apache.jsp.main_jsp._jspService(main_jsp.java:65) org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:68) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265) javax.servlet.http.HttpServlet.service(HttpServlet.java:722) ....

使用 try…catch块

如果您想要将异常处理放在一个页面中，并且对不同的异常进行不同的处理，那么您就需要使用try…catch块了。

接下来的这个例子显示了如何使用try…catch块，将这些代码放在main.jsp中:

```
<html>
<head>
   <title>Try...Catch Example</title>
</head>
<body>
<%
   try{
      int i = 1;
      i = i / 0;
      out.println("The answer is " + i);
   }
   catch (Exception e){
      out.println("An exception occurred: " + e.getMessage());
   }
%>
</body>
</html>
```

试着访问main.jsp，它将会产生如下结果:

```
An exception occurred: / by zero
```

JSP 调试

要测试/调试一个JSP或servlet程序总是那么的难。JSP和Servlets程序趋向于牵涉到大量客户端/服务器之间的交互，这很有可能会产生错误，并且很难重现出错的环境。

接下来将会给出一些小技巧和小建议，来帮助您调试程序。

使用System.out.println()

System.out.println()可以很方便地标记一段代码是否被执行。当然，我们也可以打印出各种各样的值。此外：

- 自从System对象成为Java核心对象后，它便可以使用在任何地方而不用引入额外的类。使用范围包括Servlets，JSP，RMI，EJB's，Beans， 类和独立应用。
- 与在断点处停止运行相比，用System.out进行输出不会对应用程序的运行流程造成重大的影响，这个特点在定时机制非常重要的应用程序中就显得非常有用了。

接下来给出了使用System.out.println()的语法：

```
System.out.println("Debugging message");
```

这是一个使用System.out.print()的简单例子：

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title>System.out.println</title></head>
<body>
<c:forEach var="counter" begin="1" end="10" step="1" >
   <c:out value="${counter-5}"/></br>
   <% System.out.println( "counter= " +
                     pageContext.findAttribute("counter") ); %>
</c:forEach>
</body>
</html>
```

现在，如果运行上面的例子的话，它将会产生如下的结果：

```
-4
-3
-2
-1
0
1
2
3
4
5
```

如果使用的是Tomcat服务器，您就能够在logs目录下的stdout.log文件中发现多出了如下内容：

```
counter=1
counter=2
counter=3
counter=4
counter=5
counter=6
counter=7
counter=8
counter=9
counter=10
```

使用这种方法可以将变量和其它的信息输出至系统日志中，用来分析并找出造成问题的深层次原因。

使用JDB Logger

J2SE日志框架可为任何运行在JVM中的类提供日志记录服务。因此我们可以利用这个框架来记录任何信息。

让我们来重写以上代码，使用JDK中的 logger API：

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page import="java.util.logging.Logger" %>

<html>
<head><title>Logger.info</title></head>
<body>
<% Logger logger=Logger.getLogger(this.getClass().getName());%>

<c:forEach var="counter" begin="1" end="10" step="1" >
   <c:set var="myCount" value="${counter-5}" />
   <c:out value="${myCount}"/></br>
   <% String message = "counter="
                  + pageContext.findAttribute("counter")
                  + " myCount="
                  + pageContext.findAttribute("myCount");
                  logger.info( message );
   %>
</c:forEach>
</body>
</html>
```

它的运行结果与先前的类似，但是，它可以获得额外的信息输出至stdout.log文件中。在这我们使用了logger中的info方法。下面我们给出stdout.log文件中的一个快照：

```
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=1 myCount=-4
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=2 myCount=-3
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=3 myCount=-2
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=4 myCount=-1
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=5 myCount=0
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=6 myCount=1
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=7 myCount=2
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=8 myCount=3
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=9 myCount=4
24-Sep-2013 23:31:31 org.apache.jsp.main_jsp _jspService
INFO: counter=10 myCount=5
```

消息可以使用各种优先级发送，通过使用sever()，warning()，info()，config()，fine()，finer()，finest()方法。finest()方法用来记录最好的信息，而sever()方法用来记录最严重的信息。

使用Log4J 框架来将消息记录在不同的文件中，这些消息基于严重程度和重要性来进行分类。

调试工具

NetBeans是树形结构，是开源的Java综合开发环境，支持开发独立的Java应用程序和网络应用程序，同时也支持JSP调试。

NetBeans支持如下几个基本的调试功能：

- 断点
- 单步跟踪
- 观察点

详细的信息可以查看NetBeans使用手册。

使用JDB Debugger

可以在JSP和servlets中使用jdb命令来进行调试，就像调试普通的应用程序一样。

通常，我们直接调试sun.servlet.http.HttpServer 对象来查看HttpServer在响应HTTP请求时执行JSP/Servlets的情况。这与调试applets非常相似。不同之处在于，applets程序实际调试的是sun.applet.AppletViewer。

大部分调试器在调试applets时都能够自动忽略掉一些细节，因为它知道如何调试applets。如果想要将调试对象转移到JSP身上，就需要做好以下两点：

- 设置调试器的classpath，让它能够找到sun.servlet.http.Http-Server 和相关的类。
- 设置调试器的classpath，让它能够找到您的JSP文件和相关的类。

设置好classpath后，开始调试sun.servlet.http.Http-Server 。您可以在JSP文件的任意地方设置断点，只要你喜欢，然后使用浏览器发送一个请求给服务器就应该可以看见程序停在了断点处。

使用注释

程序中的注释在很多方面都对程序的调试起到一定的帮助作用。注释可以用在调试程序的很多方面中。

JSP使用Java注释。如果一个BUG消失了，就请仔细查看您刚注释过的代码，通常都能找出原因。

客户端和服务器的头模块

有时候，当JSP没有按照预定的方式运行时，查看未加工的HTTP请求和响应也是很有用的。如果对HTTP的结构很熟悉的话，您可以直接观察request和response然后看看这些头模块到底怎么了。

重要调试技巧

这里我们再透露两个调试JSP的小技巧：

- 使用浏览器显示原始的页面内容，用来区分是否是格式问题。这个选项通常在View菜单下。
- 确保浏览器在强制重新载入页面时没有捕获先前的request输出。若使用的是Netscape Navigator浏览器，则用Shift-Reload；若使用的是IE浏览器，则用Shift-Refresh。

JSP 国际化

在开始前，需要解释几个重要的概念：

- 国际化（i18n）：表明一个页面根据访问者的语言或国家来呈现不同的翻译版本。
- 本地化（l10n）：向网站添加资源，以使它适应不同的地区和文化。比如网站的印度语版本。
- 区域：这是一个特定的区域或文化，通常认为是一个语言标志和国家标志通过下划线连接起来。比如"en_US"代表美国英语地区。

如果想要建立一个全球化的网站，就需要关心一系列项目。本章将会详细告诉您如何处理国际化问题，并给出了一些例子来加深理解。

JSP容器能够根据request的locale属性来提供正确地页面版本。接下来给出了如何通过request对象来获得Locale对象的语法：

```
java.util.Locale request.getLocale()
```

检测Locale

下表列举出了Locale对象中比较重要的方法，用于检测request对象的地区，语言，和区域。所有这些方法都会在浏览器中显示国家名称和语言名称：

方法	描述
String getCountry()	返回国家/地区码的英文大写，或 ISO 3166 2-letter 格式的区域
String getDisplayCountry()	返回要显示给用户的国家名称
String getLanguage()	返回语言码的英文小写，或ISO 639 格式的区域
String getDisplayLanguage()	返回要给用户看的语言名称
String getISO3Country()	返回国家名称的3字母缩写
String getISO3Language()	返回语言名称的3字母缩写

实例演示

这个例子告诉我们如何在JSP中显示语言和国家：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%
   //获取客户端本地化信息
   Locale locale = request.getLocale();
   String language = locale.getLanguage();
   String country = locale.getCountry();
%>
<html>
<head>
<title>Detecting Locale</title>
</head>
<body>
<center>
<h1>Detecting Locale</h1>
</center>
<p align="center">
<%
   out.println("Language : " + language  + "<br />");
   out.println("Country  : " + country   + "<br />");
%>
</p>
</body>
</html>
```

语言设置

JSP可以使用西欧语言来输出一个页面，比如英语，西班牙语，德语，法语，意大利语等等。由此可见，设置Content-Language信息头来正确显示所有字符是很重要的。

第二点就是，需要使用HTML字符实体来显示特殊字符，比如"ñ" 代表的是"?"，"¡"代表的是"?" ：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%
    // Set response content type
    response.setContentType("text/html");
    // Set spanish language code.
    response.setHeader("Content-Language", "es");
    String title = "En Espa?ol";

%>
<html>
<head>
<title><%  out.print(title); %></title>
</head>
<body>
<center>
<h1><%  out.print(title); %></h1>
</center>
<div align="center">
<p>En Espa?ol</p>
<p>?Hola Mundo!</p>
</div>
</body>
</html>
```

区域特定日期

可以使用java.text.DateFormat类和它的静态方法getDateTimeInstance()来格式化日期和时间。接下来的这个例子显示了如何根据指定的区域来格式化日期和时间：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.DateFormat,java.util.Date" %>

<%
    String title = "Locale Specific Dates";
    //Get the client's Locale
    Locale locale = request.getLocale( );
    String date = DateFormat.getDateTimeInstance(
                                  DateFormat.FULL,
                                  DateFormat.SHORT,
                                  locale).format(new Date( ));
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Local Date: <%  out.print(date); %></p>
</div>
</body>
</html>
```

区域特定货币

可以使用java.text.NumberFormat类和它的静态方法getCurrencyInstance()来格式化数字。比如在区域特定货币中的long型和double型。接下来的例子显示了如何根据指定的区域来格式化货币：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.NumberFormat,java.util.Date" %>

<%
    String title = "Locale Specific Currency";
    //Get the client's Locale
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getCurrencyInstance(locale);
    String formattedCurr = nft.format(1000000);
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Formatted Currency: <%  out.print(formattedCurr); %></p>
</div>
</body>
</html>
```

区域特定百分比

可以使用java.text.NumberFormat类和它的静态方法getPercentInstance()来格式化百分比。接下来的例子告诉我们如何根据指定的区域来格式化百分比：

```
<%@ page import="java.io.*,java.util.Locale" %>
<%@ page import="javax.servlet.*,javax.servlet.http.* "%>
<%@ page import="java.text.NumberFormat,java.util.Date" %>

<%
    String title = "Locale Specific Percentage";
    //Get the client's Locale
    Locale locale = request.getLocale( );
    NumberFormat nft = NumberFormat.getPercentInstance(locale);
    String formattedPerc = nft.format(0.51);
%>
<html>
<head>
<title><% out.print(title); %></title>
</head>
<body>
<center>
<h1><% out.print(title); %></h1>
</center>
<div align="center">
<p>Formatted Percentage: <%  out.print(formattedPerc); %></p>
</div>
</body>
</html>
```

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。